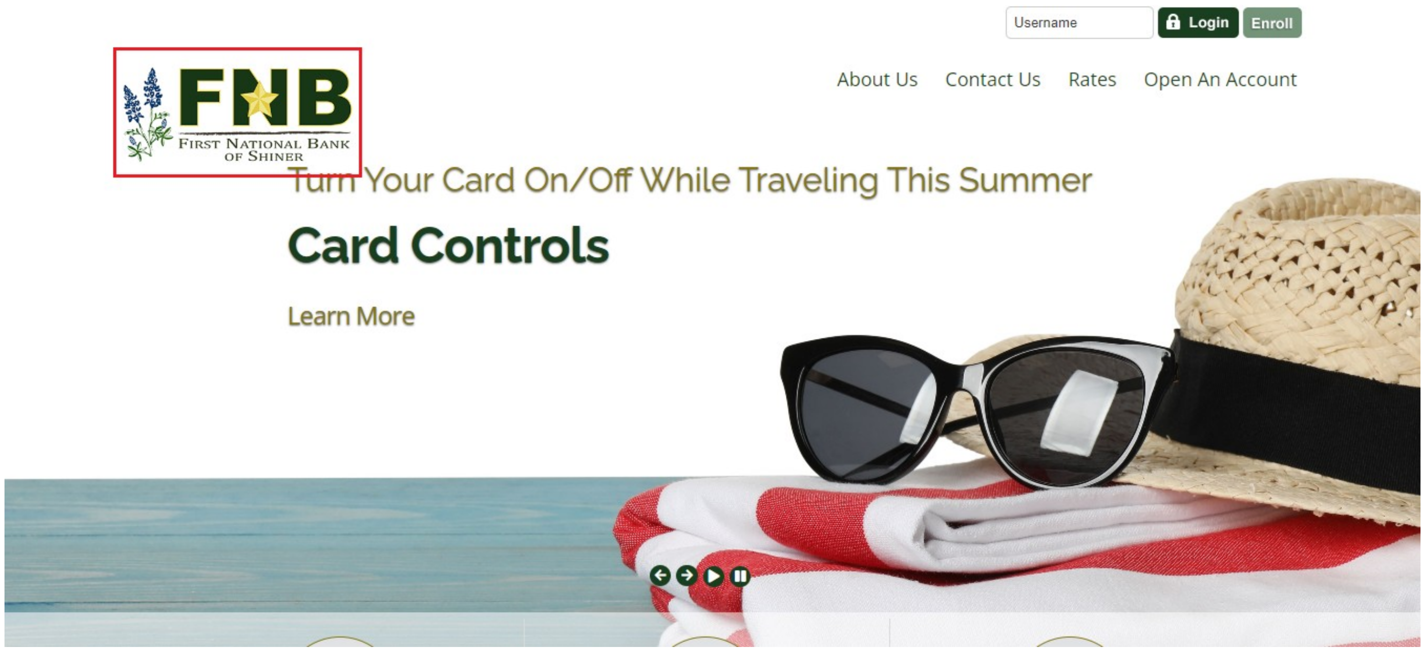
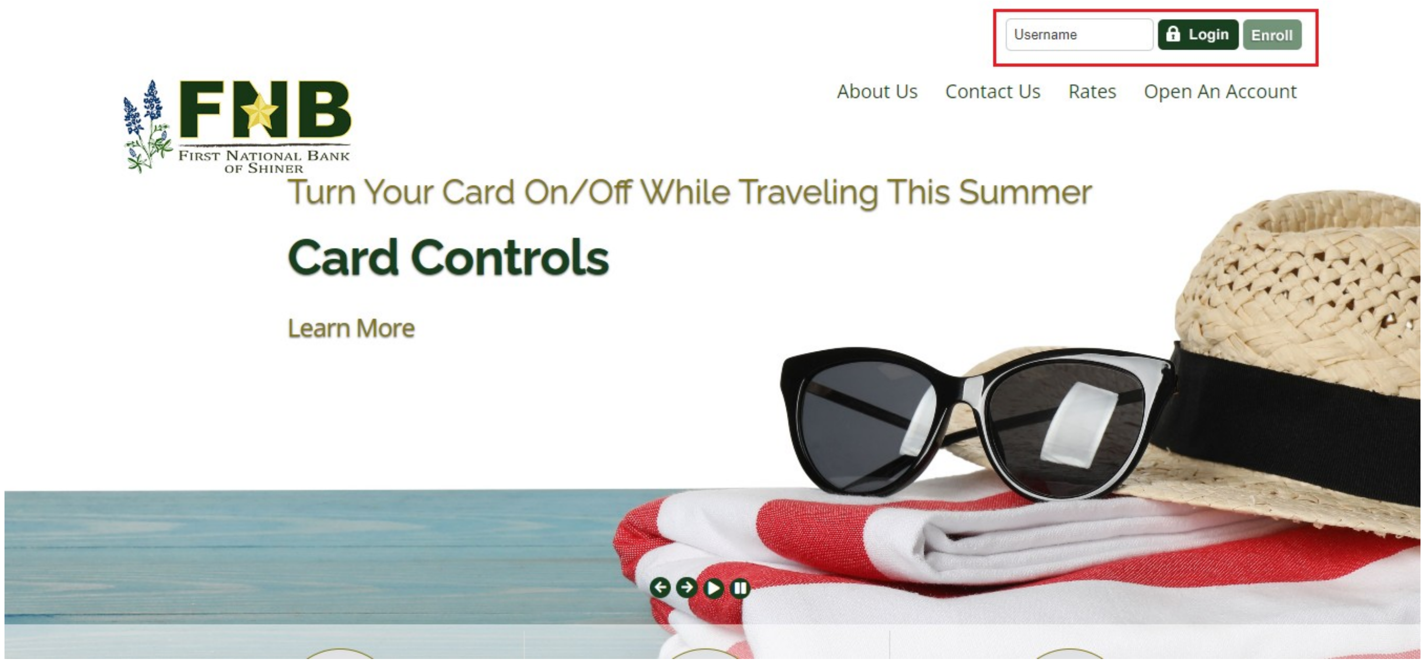
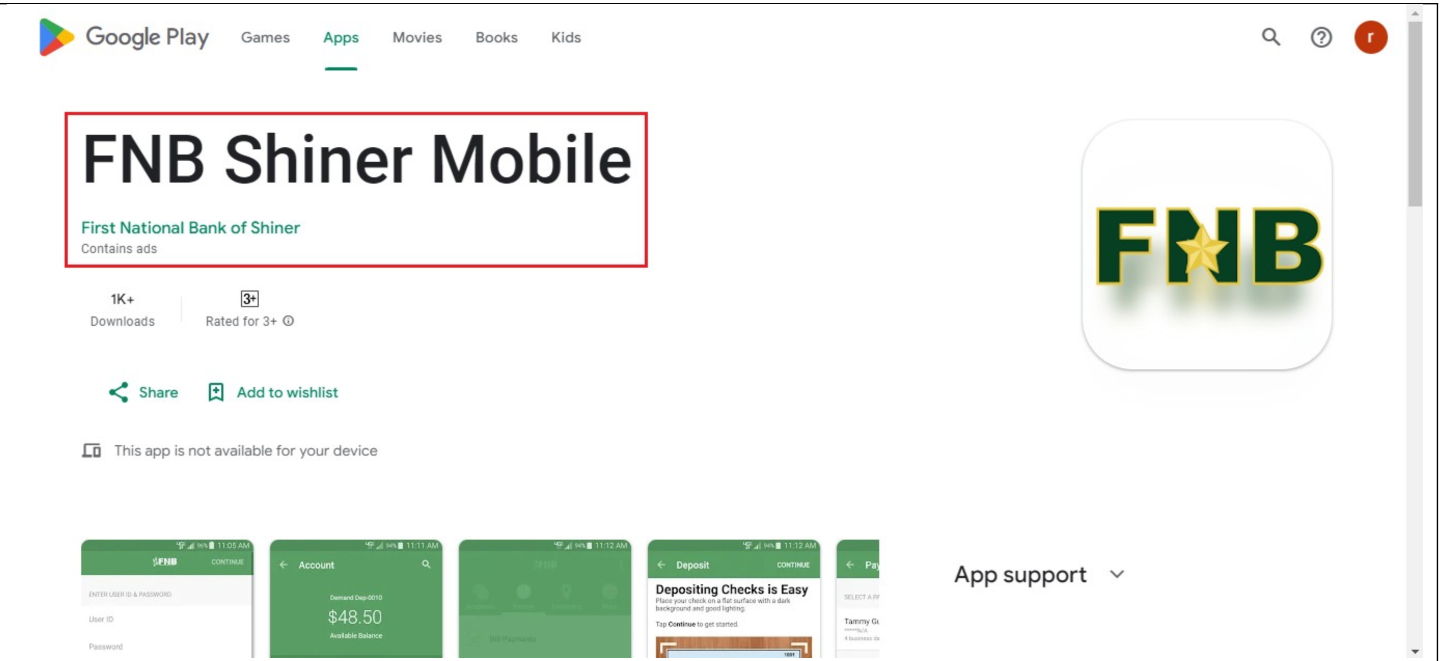
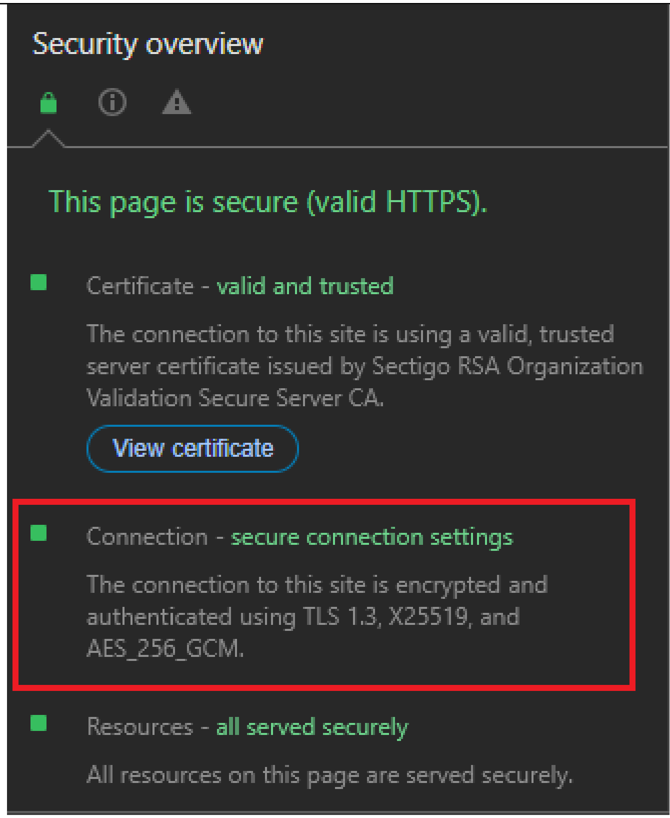
Exhibit 2

US7203844B1	First National Bank of Shiner's website fnbshiner.com ("The accused instrumentality")
1. A method for a recursive security protocol for protecting digital content, comprising:	The accused instrumentality practices a method for a recursive security protocol (e.g., TLS 1.3 security protocol) for protecting digital content (e.g., digital certificate related to the accused instrumentality). The accused instrumentality utilizes TLS 1.3 security protocol (hereinafter "the standard") for communicating content such as digital certificate, application data, etc., with a client. The standard provides a two-level encryption security. It encrypts a plaintext with a first encryption technique and generates a ciphertext. Further, it encrypts the ciphertext with a second encryption technique i.e., recursive encryption security.  https://www.fnbshiner.com/

	 https://www.fnbshiner.com/



https://play.google.com/store/apps/details?id=com.mfoundry.mb.android.mb_113106833&hl=en_US

<table>
<tr>
<td></td>
<td>


https://www.fnbshiner.com/
</td>
</tr>
</table>

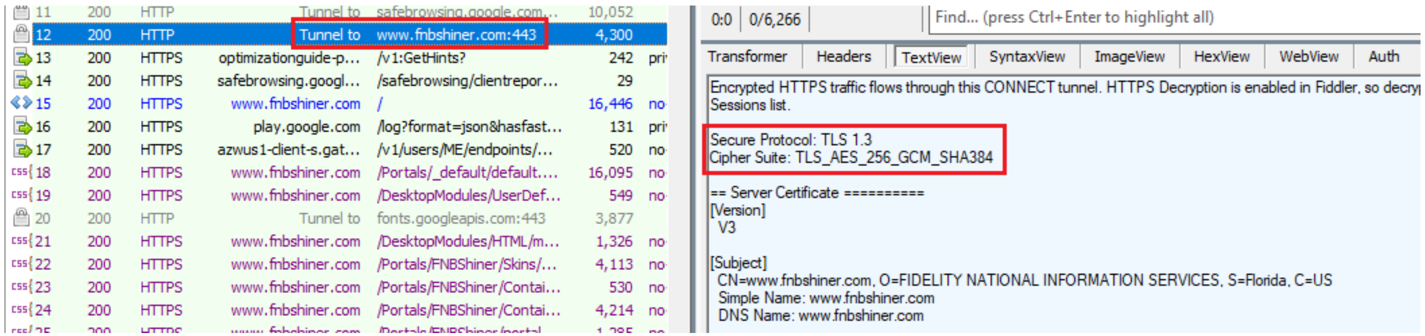
The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

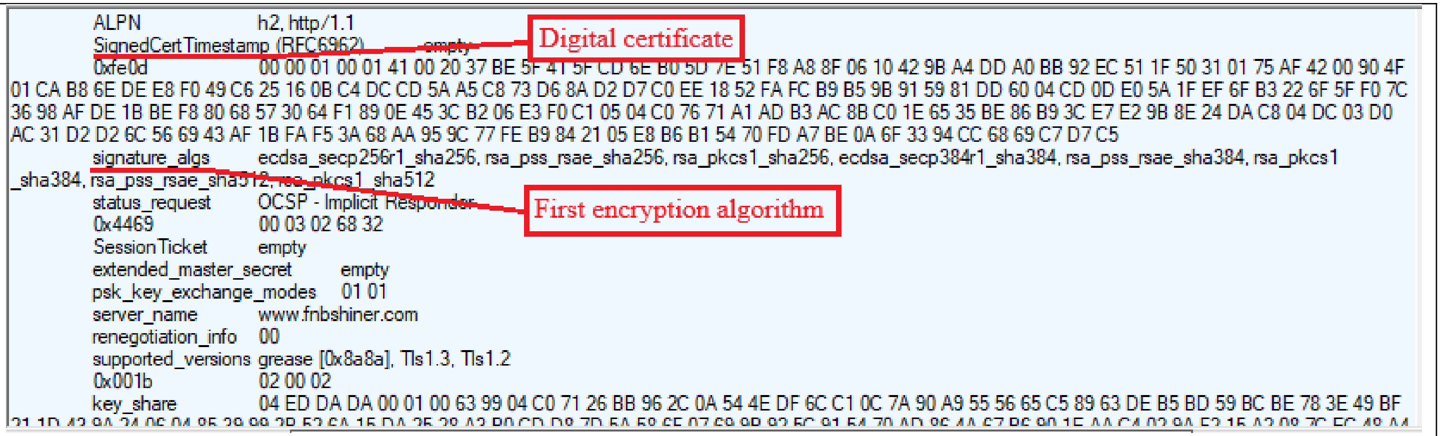
This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

https://datatracker.ietf.org/doc/html/rfc8446

As shown below, the accused instrumentality utilizes a two-level algorithm security. It utilizes the SHA256RSA encryption algorithm as a first encryption algorithm i.e., signature encryption algorithm and the TLS_AES_256_GCM_SHA384 encryption algorithm as a second encryption algorithm i.e., AEAD encryption algorithm.

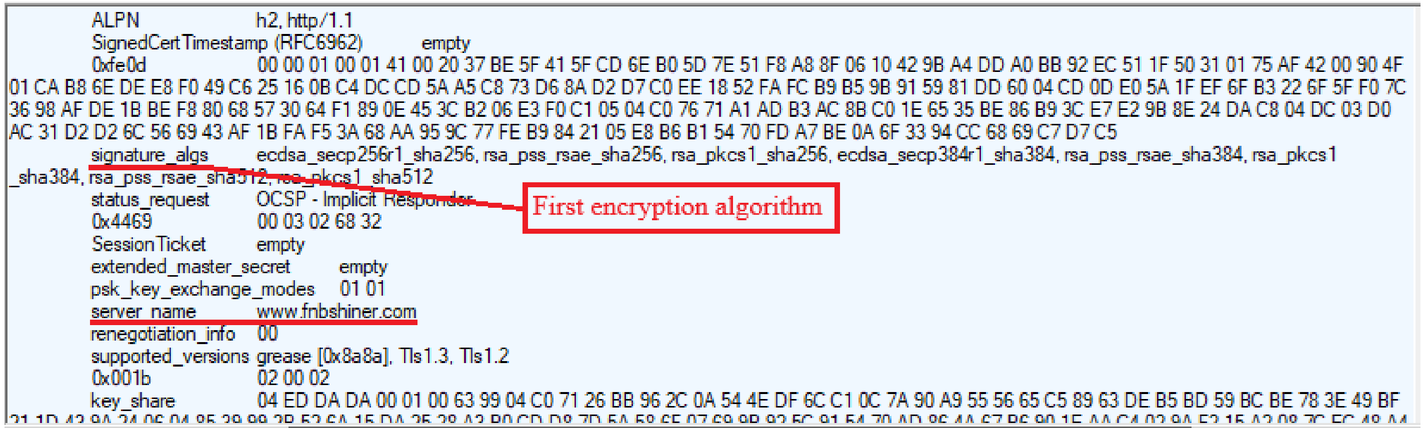


Source: Fiddler Capture



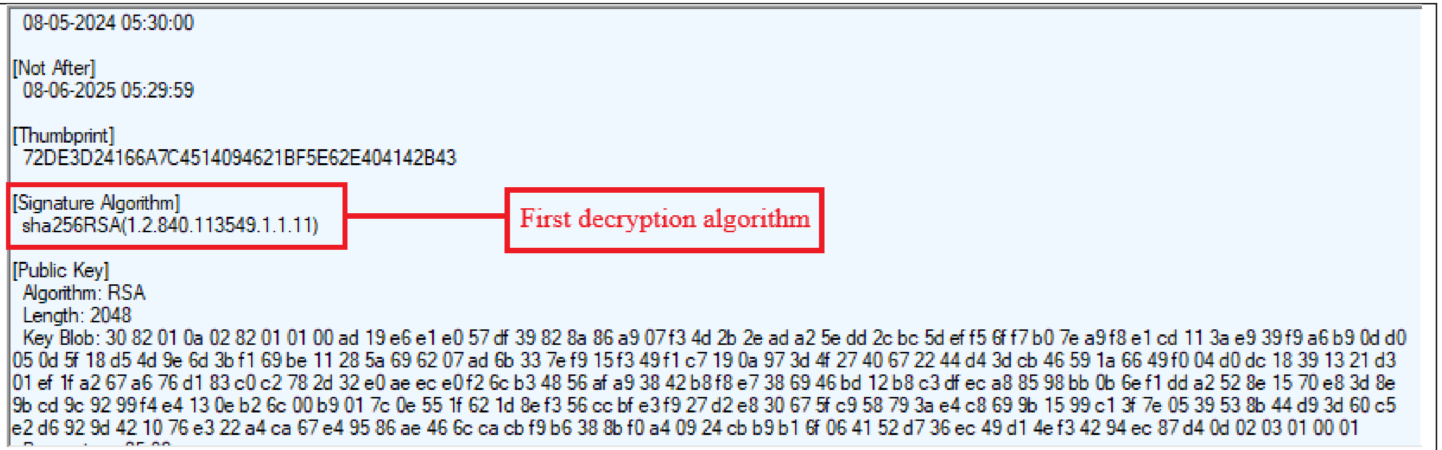
ALPN h2, http/1.1
SignedCertTimestamp (RFC6962) empty
0xfe0d 00 00 01 00 01 41 00 20 37 BE 5F 41 5F CD 6E B0 5D 7E 51 F8 A8 8F 06 10 42 9B A4 DD A0 BB 92 EC 51 1F 50 31 01 75 AF 42 00 90 4F 01 CA B8 6E DE E8 F0 49 C6 25 16 0B C4 DC CD 5A A5 C8 73 D6 8A D2 D7 C0 EE 18 52 FA FC B9 B5 9B 91 59 81 DD 60 04 CD 0D E0 5A 1F EF 6F B3 22 6F 5F F0 7C 36 98 AF DE 1B BE F8 80 68 57 30 64 F1 89 0E 45 3C B2 06 E3 F0 C1 05 04 C0 76 71 A1 AD B3 AC 8B C0 1E 65 35 BE 86 B9 3C E7 E2 9B 8E 24 DA C8 04 DC 03 D0 AC 31 D2 D2 6C 56 69 43 AF 1B FA F5 3A 68 AA 95 9C 77 FE B9 84 21 05 E8 B6 B1 54 70 FD A7 BE 0A 6F 33 94 CC 68 69 C7 D7 C5
signature_algs ecdsa_secp256r1_sha256, rsa_pss_rsae_sha256, rsa_pkcs1_sha256, ecdsa_secp384r1_sha384, rsa_pss_rsae_sha384, rsa_pkcs1_sha384, rsa_pss_rsae_sha512, rsa_pkcs1_sha512
status_request OCSP - Implicit Responder
0x4469 00 03 02 68 32
SessionTicket empty
extended_master_secret empty
psk_key_exchange_modes 01 01
server_name www.fnbshiner.com
renegotiation_info 00
supported_versions grease [0x8a8a], Tls1.3, Tls1.2
0x001b 02 00 02
key_share 04 ED DA DA 00 01 00 63 99 04 C0 71 26 BB 96 2C 0A 54 4E DF 6C C1 0C 7A 90 A9 55 56 65 C5 89 63 DE B5 BD 59 BC BE 78 3E 49 BF

Source: Fiddler Capture

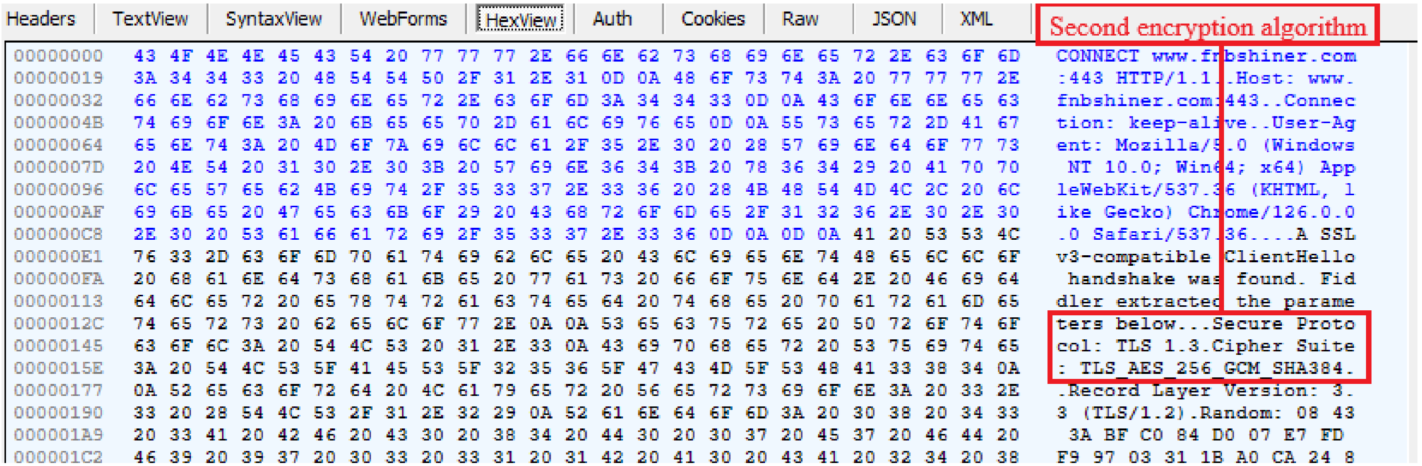


ALPN h2, http/1.1
SignedCertTimestamp (RFC6962) empty
0xfe0d 00 00 01 00 01 41 00 20 37 BE 5F 41 5F CD 6E B0 5D 7E 51 F8 A8 8F 06 10 42 9B A4 DD A0 BB 92 EC 51 1F 50 31 01 75 AF 42 00 90 4F 01 CA B8 6E DE E8 F0 49 C6 25 16 0B C4 DC CD 5A A5 C8 73 D6 8A D2 D7 C0 EE 18 52 FA FC B9 B5 9B 91 59 81 DD 60 04 CD 0D E0 5A 1F EF 6F B3 22 6F 5F F0 7C 36 98 AF DE 1B BE F8 80 68 57 30 64 F1 89 0E 45 3C B2 06 E3 F0 C1 05 04 C0 76 71 A1 AD B3 AC 8B C0 1E 65 35 BE 86 B9 3C E7 E2 9B 8E 24 DA C8 04 DC 03 D0 AC 31 D2 D2 6C 56 69 43 AF 1B FA F5 3A 68 AA 95 9C 77 FE B9 84 21 05 E8 B6 B1 54 70 FD A7 BE 0A 6F 33 94 CC 68 69 C7 D7 C5
signature_algs ecdsa_secp256r1_sha256, rsa_pss_rsae_sha256, rsa_pkcs1_sha256, ecdsa_secp384r1_sha384, rsa_pss_rsae_sha384, rsa_pkcs1_sha384, rsa_pss_rsae_sha512, rsa_pkcs1_sha512
status_request OCSP - Implicit Responder
0x4469 00 03 02 68 32
SessionTicket empty
extended_master_secret empty
psk_key_exchange_modes 01 01
server_name www.fnbshiner.com
renegotiation_info 00
supported_versions grease [0x8a8a], Tls1.3, Tls1.2
0x001b 02 00 02
key_share 04 ED DA DA 00 01 00 63 99 04 C0 71 26 BB 96 2C 0A 54 4E DF 6C C1 0C 7A 90 A9 55 56 65 C5 89 63 DE B5 BD 59 BC BE 78 3E 49 BF

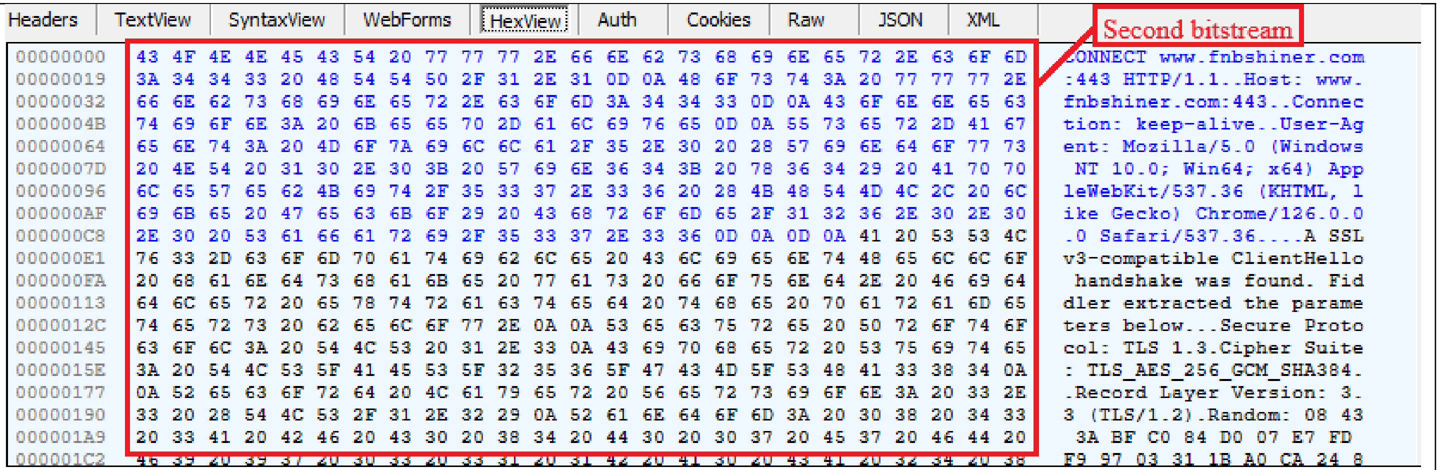
Source: Fiddler Capture



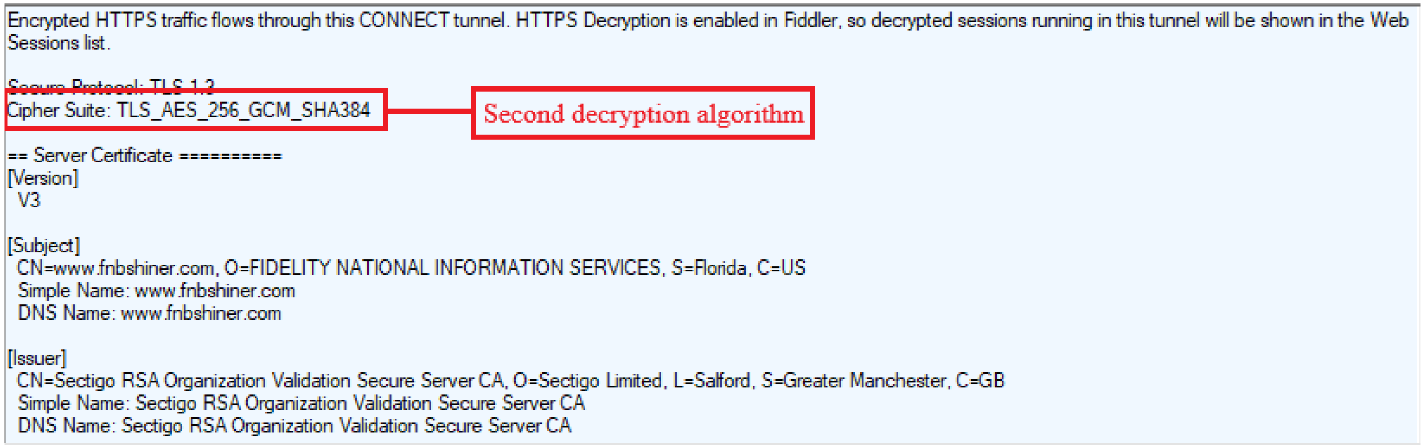
Source: Fiddler Capture



Source: Fiddler Capture



Source: Fiddler Capture



Source: Fiddler Capture

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for

	encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm. As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server. **5. Record Protocol** The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients. TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see Appendix D.4). https://datatracker.ietf.org/doc/html/rfc8446#section-1

	2. Protocol Overview Negotiating encryption algorithm The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic. https://datatracker.ietf.org/doc/html/rfc8446 TLS consists of two primary components: - A handshake protocol (Section 4) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack. Negotiating encryption algos - A record protocol (Section 5) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys. https://datatracker.ietf.org/doc/html/rfc8446

	All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **4.4. Authentication Messages** As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [sender]_handshake_traffic_secret. https://datatracker.ietf.org/doc/html/rfc8446#section-1

Figure 1 below shows the basic full TLS handshake:

Digital Content

```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.1.1. Cryptographic Negotiation In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello: - A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports. Second encryption - A "supported_groups" (Section 4.2.7) extension which indicates the (EC)DHE groups which the client supports and a "key_share" (Section 4.2.8) extension which contains (EC)DHE shares for some or all of these groups. - A "signature_algorithms" (Section 4.2.3) extension which indicates the signature algorithms which the client can accept. A "signature_algorithms_cert" extension (Section 4.2.3) may also be added to indicate certificate-specific signature algorithms. First encryption - A "pre_shared_key" (Section 4.2.11) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" (Section 4.2.9) extension which indicates the key exchange modes that may be used with PSKs. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2. Certificate This message conveys the endpoint's certificate chain to the peer. The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK). The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2.3. Client Certificate Selection The following rules apply to certificates sent by the client: - The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]). - If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs. - The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS. - If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.3. Certificate Verify This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message. Structure of this message: `struct {` `   SignatureScheme algorithm;` `   opaque signature<0..2^16-1>;` `} CertificateVerify;` The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely: https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	4.3.2. Certificate Request A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions. Structure of this message: `struct {` `opaque certificate_request_context<0..2^8-1>;` `Extension extensions<2..2^16-1>;` `} CertificateRequest;` https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages. extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions. https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message. In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.2.3. Signature Algorithms TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2). The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	The "extension_data" field of these extensions contains a SignatureSchemeList value: `enum {` `    /* RSASSA-PKCS1-v1_5 algorithms */` `    rsa_pkcs1_sha256(0x0401),` `    rsa_pkcs1_sha384(0x0501),` `    rsa_pkcs1_sha512(0x0601),` `    /* ECDSA algorithms */` `    ecdsa_secp256r1_sha256(0x0403),` `    ecdsa_secp384r1_sha384(0x0503),` `    ecdsa_secp521r1_sha512(0x0603),` `    /* RSASSA-PSS algorithms with public key OID rsaEncryption */` `    rsa_pss_rsae_sha256(0x0804),` `    rsa_pss_rsae_sha384(0x0805),` `    rsa_pss_rsae_sha512(0x0806),` `    /* EdDSA algorithms */` `    ed25519(0x0807),` `    ed448(0x0808),` `    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */` `    rsa_pss_pss_sha256(0x0809),` `    rsa_pss_pss_sha384(0x080a),` `    rsa_pss_pss_sha512(0x080b),` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	Introduction The primary goal of TLS is to provide a secure channel between two communicating peers; the only requirement from the underlying transport is a reliable, in-order data stream. Specifically, the secure channel should provide the following properties: First encryption - Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated. Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], the Elliptic Curve Digital Signature Algorithm (ECDSA) [ECDSA], or the Edwards-Curve Digital Signature Algorithm (EdDSA) [RFC8032]) or a symmetric pre-shared key (PSK). - Confidentiality: Data sent over the channel after establishment is only visible to the endpoints. TLS does not hide the length of the data it transmits, though endpoints are able to pad TLS records in order to obscure lengths and improve protection against traffic analysis techniques. - Integrity: Data sent over the channel after establishment cannot be modified by attackers without detection. https://datatracker.ietf.org/doc/html/rfc8446

	5.1. Record Layer The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^14 bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2. Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that: - Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **5.2. Record Payload Protection** The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding. https://datatracker.ietf.org/doc/html/rfc8446#section-1

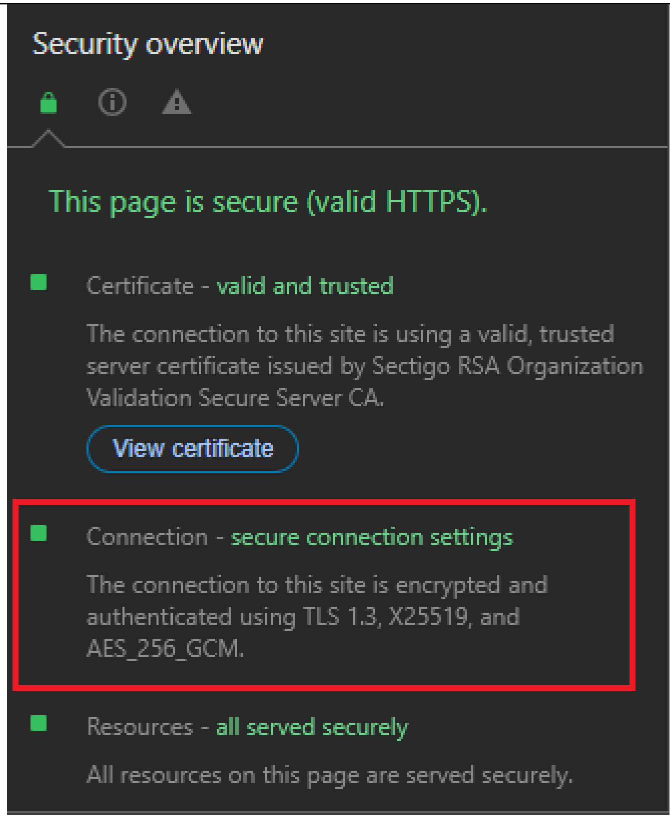
	AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see Section 5.3), and the additional data input is the record header. I.e., `additional_data = TLSCiphertext.opaque_type ‖` `TLSCiphertext.legacy_record_version ‖` `TLSCiphertext.length` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case. https://datatracker.ietf.org/doc/html/rfc5116

	2.1. Authenticated Encryption The authenticated encryption operation has four inputs, each of which is an octet string: A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom. A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in Section 3.2, and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces. A plaintext P, which contains the data to be encrypted and authenticated. The associated data A, which contains the data to be authenticated, but not encrypted. https://datatracker.ietf.org/doc/html/rfc5116

	The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1 Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [RFC5116], Section 4). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows: https://datatracker.ietf.org/doc/html/rfc8446#section-1

	This specification defines the following cipher suites for use with TLS 1.3. ``` +------------------------------+-------------+ | Description | Value | +------------------------------+-------------+ | TLS_AES_128_GCM_SHA256 | {0x13,0x01} | | | | | TLS_AES_256_GCM_SHA384 | {0x13,0x02} | | | | | TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} | | | | | TLS_AES_128_CCM_SHA256 | {0x13,0x04} | | | | | TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} | +------------------------------+-------------+ ``` https://datatracker.ietf.org/doc/html/rfc8446#section-1
encrypting a bitstream with a first encryption algorithm;	The standard practices encrypting a bitstream (e.g., bitstream of digital certificate) with a first encryption algorithm (e.g., signature encryption algorithm i.e., SHA256RSA encryption algorithm). The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA encryption algorithm) and generates a ciphertext.



https://www.fnbshiner.com/

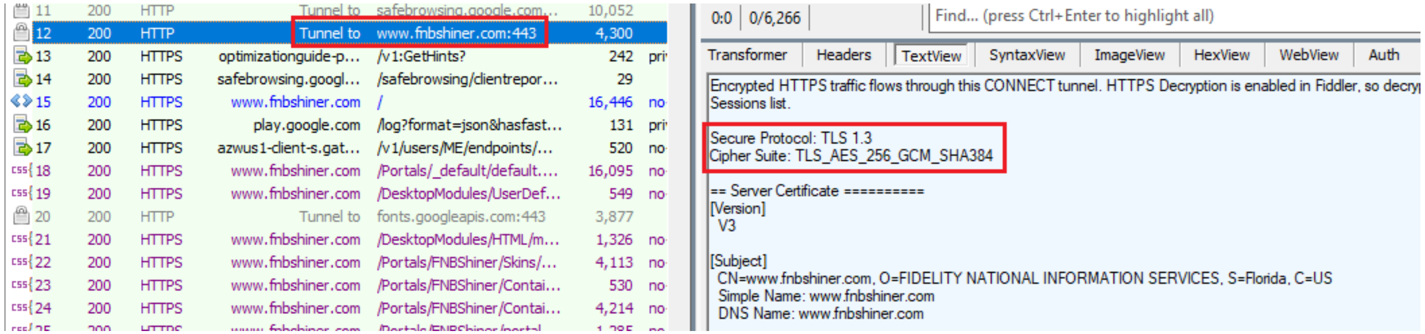
The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

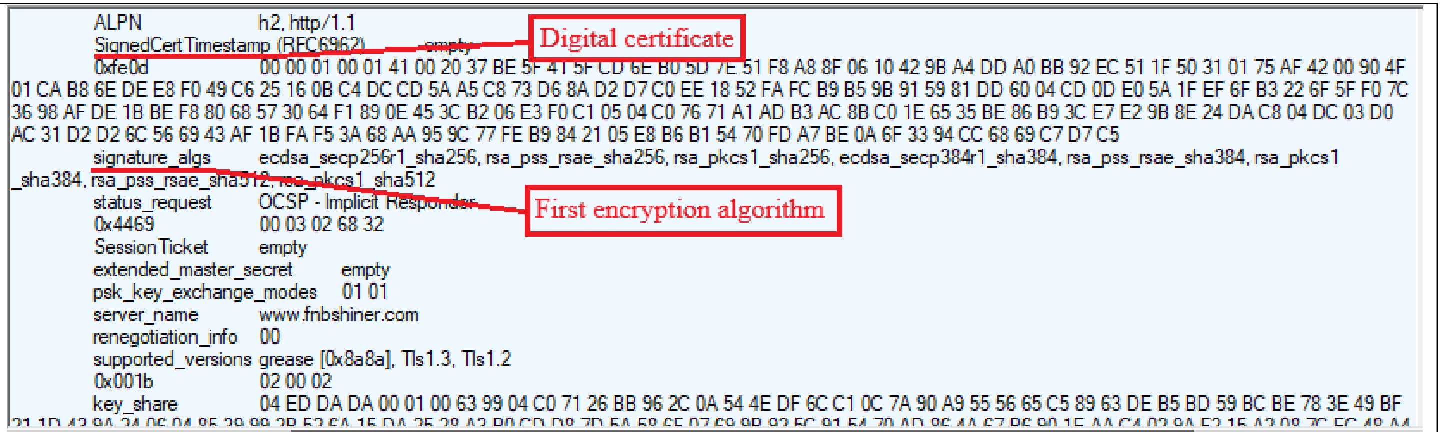
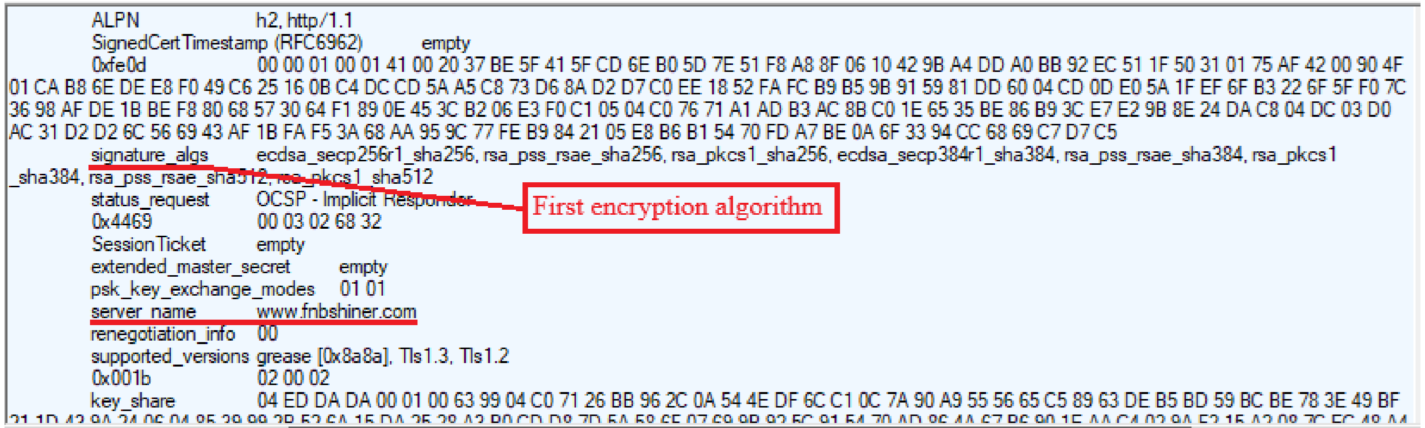
This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

https://datatracker.ietf.org/doc/html/rfc8446

As shown below, the accused instrumentality discloses the signature encryption algorithm.



Source: Fiddler Capture

	 *Source: Fiddler Capture*  *Source: Fiddler Capture* The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are

	encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm. As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server. **5. Record Protocol** The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients. TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see Appendix D.4). https://datatracker.ietf.org/doc/html/rfc8446#section-1

	2. Protocol Overview Negotiating encryption algorithm The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic. https://datatracker.ietf.org/doc/html/rfc8446 TLS consists of two primary components: - A handshake protocol (Section 4) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack. Negotiating encryption algos - A record protocol (Section 5) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys. https://datatracker.ietf.org/doc/html/rfc8446

	All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **4.4. Authentication Messages** As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [sender]_handshake_traffic_secret. https://datatracker.ietf.org/doc/html/rfc8446#section-1

Figure 1 below shows the basic full TLS handshake:

```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

Digital Content

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.1.1. Cryptographic Negotiation In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello: - A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports. - A "supported_groups" (Section 4.2.7) extension which indicates the (EC)DHE groups which the client supports and a "key_share" (Section 4.2.8) extension which contains (EC)DHE shares for some or all of these groups. First encryption - A "signature_algorithms" (Section 4.2.3) extension which indicates the signature algorithms which the client can accept. A "signature_algorithms_cert" extension (Section 4.2.3) may also be added to indicate certificate-specific signature algorithms. - A "pre_shared_key" (Section 4.2.11) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" (Section 4.2.9) extension which indicates the key exchange modes that may be used with PSKs. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2. Certificate This message conveys the endpoint's certificate chain to the peer. The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK). The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2.3. Client Certificate Selection The following rules apply to certificates sent by the client: - The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]). - If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs. - The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS. - If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.3. Certificate Verify This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message. Structure of this message: `struct {` `   SignatureScheme algorithm;` `   opaque signature<0..2^16-1>;` `} CertificateVerify;` The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely: https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	4.3.2. Certificate Request A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions. Structure of this message: `struct {` `opaque certificate_request_context<0..2^8-1>;` `Extension extensions<2..2^16-1>;` `} CertificateRequest;` https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

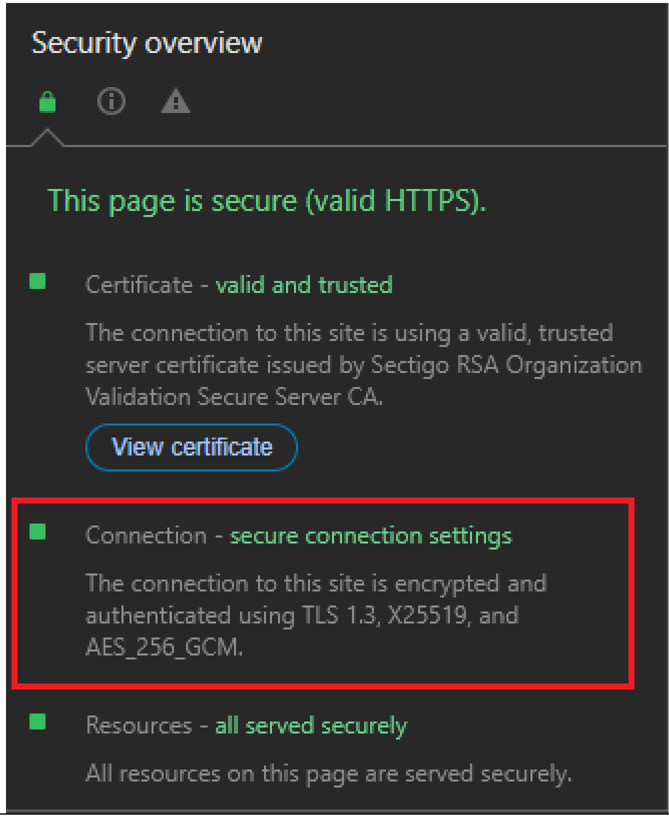
	certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages. extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions. https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message. In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.2.3. Signature Algorithms TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2). The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	The "extension_data" field of these extensions contains a SignatureSchemeList value: `enum {` `/* RSASSA-PKCS1-v1_5 algorithms */` `rsa_pkcs1_sha256(0x0401),` `rsa_pkcs1_sha384(0x0501),` `rsa_pkcs1_sha512(0x0601),` `/* ECDSA algorithms */` `ecdsa_secp256r1_sha256(0x0403),` `ecdsa_secp384r1_sha384(0x0503),` `ecdsa_secp521r1_sha512(0x0603),` `/* RSASSA-PSS algorithms with public key OID rsaEncryption */` `rsa_pss_rsae_sha256(0x0804),` `rsa_pss_rsae_sha384(0x0805),` `rsa_pss_rsae_sha512(0x0806),` `/* EdDSA algorithms */` `ed25519(0x0807),` `ed448(0x0808),` `/* RSASSA-PSS algorithms with public key OID RSASSA-PSS */` `rsa_pss_pss_sha256(0x0809),` `rsa_pss_pss_sha384(0x080a),` `rsa_pss_pss_sha512(0x080b),` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	Introduction The primary goal of TLS is to provide a secure channel between two communicating peers; the only requirement from the underlying transport is a reliable, in-order data stream. Specifically, the secure channel should provide the following properties: First encryption - Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated. Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], the Elliptic Curve Digital Signature Algorithm (ECDSA) [ECDSA], or the Edwards-Curve Digital Signature Algorithm (EdDSA) [RFC8032]) or a symmetric pre-shared key (PSK). - Confidentiality: Data sent over the channel after establishment is only visible to the endpoints. TLS does not hide the length of the data it transmits, though endpoints are able to pad TLS records in order to obscure lengths and improve protection against traffic analysis techniques. - Integrity: Data sent over the channel after establishment cannot be modified by attackers without detection. https://datatracker.ietf.org/doc/html/rfc8446
associating a first decryption algorithm with the encrypted bit stream;	The standard practices associating a first decryption algorithm (e.g., signature decryption algorithm i.e., SHA256RSA decryption algorithm) with the encrypted bit stream (e.g., encrypted certificate with signature encryption algorithm). The standard practices providing a two-level encryption security for data

	communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA encryption algorithm) and generates a ciphertext. The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it. The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate. 

	https://www.fnbshiner.com/

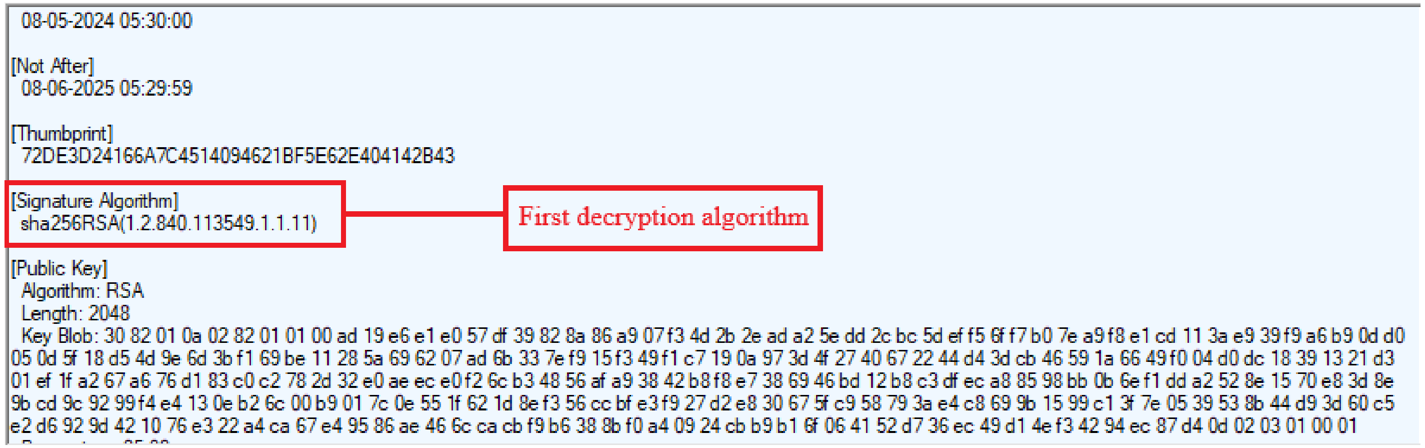
The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

https://datatracker.ietf.org/doc/html/rfc8446

As shown below, the accused instrumentality discloses the signature decryption algorithm.


08-05-2024 05:30:00

[Not After]
08-06-2025 05:29:59

[Thumbprint]
72DE3D24166A7C4514094621BF5E62E404142B43

[Signature Algorithm]
sha256RSA(1.2.840.113549.1.1.11)

First decryption algorithm

[Public Key]
Algorithm: RSA
Length: 2048
Key Blob: 30 82 01 0a 02 82 01 01 00 ad 19 e6 e1 e0 57 df 39 82 8a 86 a9 07 f3 4d 2b 2e ad a2 5e dd 2c bc 5d ef f5 6f f7 b0 7e a9 f8 e1 cd 11 3a e9 39 f9 a6 b9 0d d0
05 0d 5f 18 d5 4d 9e 6d 3b f1 69 be 11 28 5a 69 62 07 ad 6b 33 7e f9 15 f3 49 f1 c7 19 0a 97 3d 4f 27 40 67 22 44 d4 3d cb 46 59 1a 66 49 f0 04 d0 dc 18 39 13 21 d3
01 ef 1f a2 67 a6 76 d1 83 c0 c2 78 2d 32 e0 ae ec e0 f2 6c b3 48 56 af a9 38 42 b8 f8 e7 38 69 46 bd 12 b8 c3 df ec a8 85 98 bb 0b 6e f1 dd a2 52 8e 15 70 e8 3d 8e
9b cd 9c 92 99 f4 e4 13 0e b2 6c 00 b9 01 7c 0e 55 1f 62 1d 8e f3 56 cc bf e3 f9 27 d2 e8 30 67 5f c9 58 79 3a e4 c8 69 9b 15 99 c1 3f 7e 05 39 53 8b 44 d9 3d 60 c5
e2 d6 92 9d 42 10 76 e3 22 a4 ca 67 e4 95 86 ae 46 6c ca cb f9 b6 38 8b f0 a4 09 24 cb b9 b1 6f 06 41 52 d7 36 ec 49 d1 4e f3 42 94 ec 87 d4 0d 02 03 01 00 01

Source: Fiddler Capture

	OID description First decryption algorithm identifier **OID**: {iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs-1(1) sha256WithRSAEncryption(11)} (ASN.1 notation) 1.2.840.113549.1.1.11 (dot notation) /ISO/Member-Body/US/113549/1/1/11 (OID-IRI notation) **Description**: Public-Key Cryptography Standards (PKCS) #1 version 1.5 signature algorithm with Secure Hash Algorithm 256 (SHA256) and Rivest, Shamir and Adleman (RSA) encryption http://oid-info.com/get/1.2.840.113549.1.1.11 `-- When the following OIDs are used in an AlgorithmIdentifier, the` `-- parameters MUST be present and MUST be NULL.` `sha224WithRSAEncryption  OBJECT IDENTIFIER  ::=  { pkcs-1 14 }` `sha256WithRSAEncryption  OBJECT IDENTIFIER  ::=  { pkcs-1 11 }` `sha384WithRSAEncryption  OBJECT IDENTIFIER  ::=  { pkcs-1 12 }` `sha512WithRSAEncryption  OBJECT IDENTIFIER  ::=  { pkcs-1 13 }` https://www.ietf.org/rfc/rfc4055.txt

```
   Figure 1 below shows the basic full TLS handshake:

       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

Digital Content

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.3. Certificate Verify This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message. Structure of this message: `struct {` `SignatureScheme algorithm;` `opaque signature<0..2^16-1>;` `} CertificateVerify;` The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely: https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages. extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions. https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	- RSASSA-PSS signature schemes are defined in Section 4.2.3. - The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello. - The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates. https://datatracker.ietf.org/doc/html/rfc8446#section-1 The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm. As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

	5. Record Protocol The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients. TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see Appendix D.4). https://datatracker.ietf.org/doc/html/rfc8446#section-1 **2. Protocol Overview** Negotiating encryption algorithm The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic. https://datatracker.ietf.org/doc/html/rfc8446

	TLS consists of two primary components: - A handshake protocol (Section 4) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack. Negotiating encryption algos - A record protocol (Section 5) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys. https://datatracker.ietf.org/doc/html/rfc8446 All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4. Authentication Messages As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [sender]_handshake_traffic_secret. https://datatracker.ietf.org/doc/html/rfc8446#section-1

```
   Figure 1 below shows the basic full TLS handshake:

       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

Digital Content

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.1.1. Cryptographic Negotiation In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello: - A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports. - A "supported_groups" (Section 4.2.7) extension which indicates the (EC)DHE groups which the client supports and a "key_share" (Section 4.2.8) extension which contains (EC)DHE shares for some or all of these groups. First encryption - A "signature_algorithms" (Section 4.2.3) extension which indicates the signature algorithms which the client can accept. A "signature_algorithms_cert" extension (Section 4.2.3) may also be added to indicate certificate-specific signature algorithms. - A "pre_shared_key" (Section 4.2.11) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" (Section 4.2.9) extension which indicates the key exchange modes that may be used with PSKs. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2. Certificate This message conveys the endpoint's certificate chain to the peer. The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK). The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2.3. Client Certificate Selection The following rules apply to certificates sent by the client: - The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]). - If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs. - The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS. - If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5. https://datatracker.ietf.org/doc/html/rfc8446#section-1

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {
    SignatureScheme algorithm;
    opaque signature<0..2^16-1>;
} CertificateVerify;
```

First decryption algorithm information

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	4.3.2. Certificate Request A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions. Structure of this message: `struct {` `opaque certificate_request_context<0..2^8-1>;` `Extension extensions<2..2^16-1>;` `} CertificateRequest;` https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages. extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions. https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	- RSASSA-PSS signature schemes are defined in Section 4.2.3. - The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello. - The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates. https://datatracker.ietf.org/doc/html/rfc8446#section-1 If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message. In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages. https://datatracker.ietf.org/doc/html/rfc8446#section-1

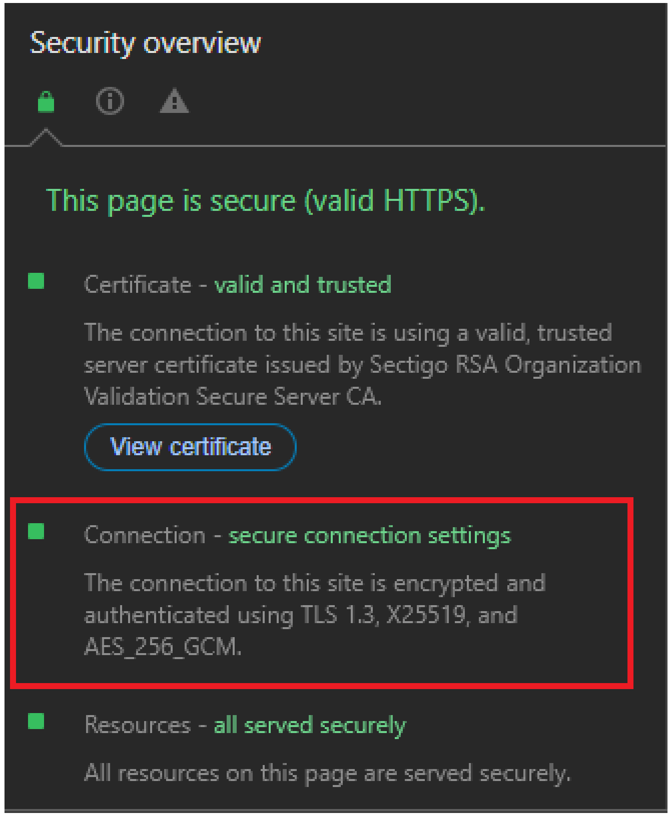
	4.2.3. Signature Algorithms TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2). The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	The "extension_data" field of these extensions contains a SignatureSchemeList value: `enum {` `/* RSASSA-PKCS1-v1_5 algorithms */` `rsa_pkcs1_sha256(0x0401),` `rsa_pkcs1_sha384(0x0501),` `rsa_pkcs1_sha512(0x0601),` `/* ECDSA algorithms */` `ecdsa_secp256r1_sha256(0x0403),` `ecdsa_secp384r1_sha384(0x0503),` `ecdsa_secp521r1_sha512(0x0603),` `/* RSASSA-PSS algorithms with public key OID rsaEncryption */` `rsa_pss_rsae_sha256(0x0804),` `rsa_pss_rsae_sha384(0x0805),` `rsa_pss_rsae_sha512(0x0806),` `/* EdDSA algorithms */` `ed25519(0x0807),` `ed448(0x0808),` `/* RSASSA-PSS algorithms with public key OID RSASSA-PSS */` `rsa_pss_pss_sha256(0x0809),` `rsa_pss_pss_sha384(0x080a),` `rsa_pss_pss_sha512(0x080b),` https://datatracker.ietf.org/doc/html/rfc8446#section-1 As shown below, the receiving party will be able to decrypt the encrypted message with the provided signature decryption algorithm information i.e., SHA-256 RSA decryption algorithm.

	We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q, compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de = 1 \pmod{\varphi(n)}$. we know that $de-1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$. Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is $\boxed{c = m^e \bmod n.}$ First encryption To decrypt, we compute $c^d \bmod n$ to obtain $c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$ The result of Exercise 3.13 tells us that $\boxed{m \equiv m^{1+k\varphi(n)} \pmod{n},}$ First decryption https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf
encrypting both the encrypted bit stream and the first decryption algorithm with a second encryption algorithm to yield a second bit stream;	The standard practices encrypting both the encrypted bit stream (e.g., encrypted digital certificate) and the first decryption algorithm (e.g., signature decryption algorithm) with a second encryption algorithm (e.g., cipher suit selected from one of the AEAD algorithms such as TLS_AES_256_GCM_SHA384, etc.) to yield a second bit stream (e.g., TLS ciphertext bitstream). The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA algorithm) and generates a ciphertext. The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it.

The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate. The standard further practices encrypting the authentication message, including the encrypted digital certification and the certificate verify message, with a second decryption algorithm i.e., AEAD algorithm such as TLS_AES_256_GCM_SHA384, etc.



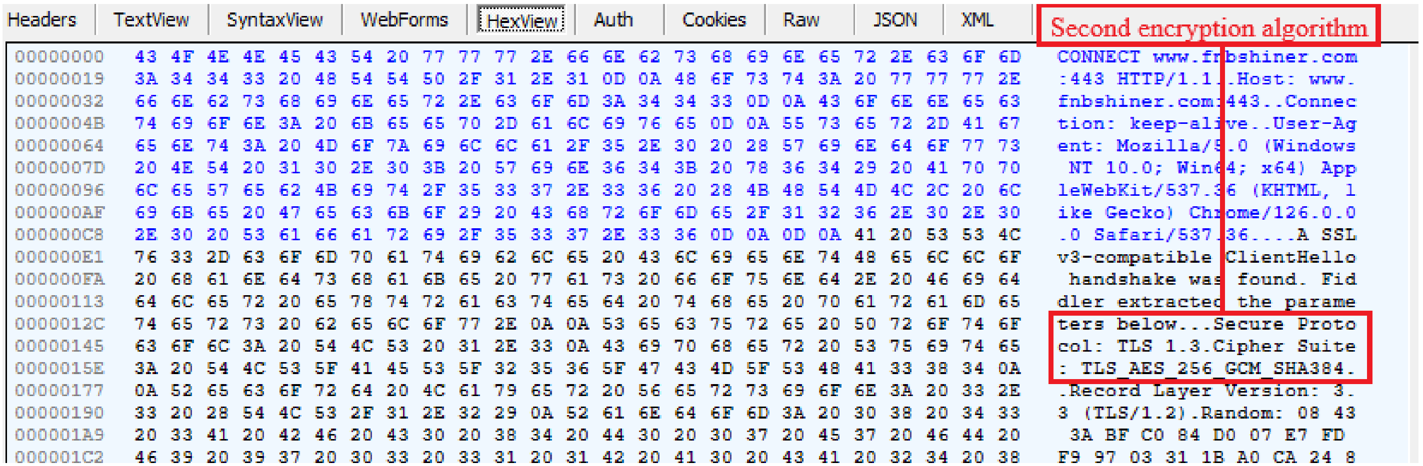
https://www.fnbshiner.com/

The Transport Layer Security (TLS) Protocol Version 1.3

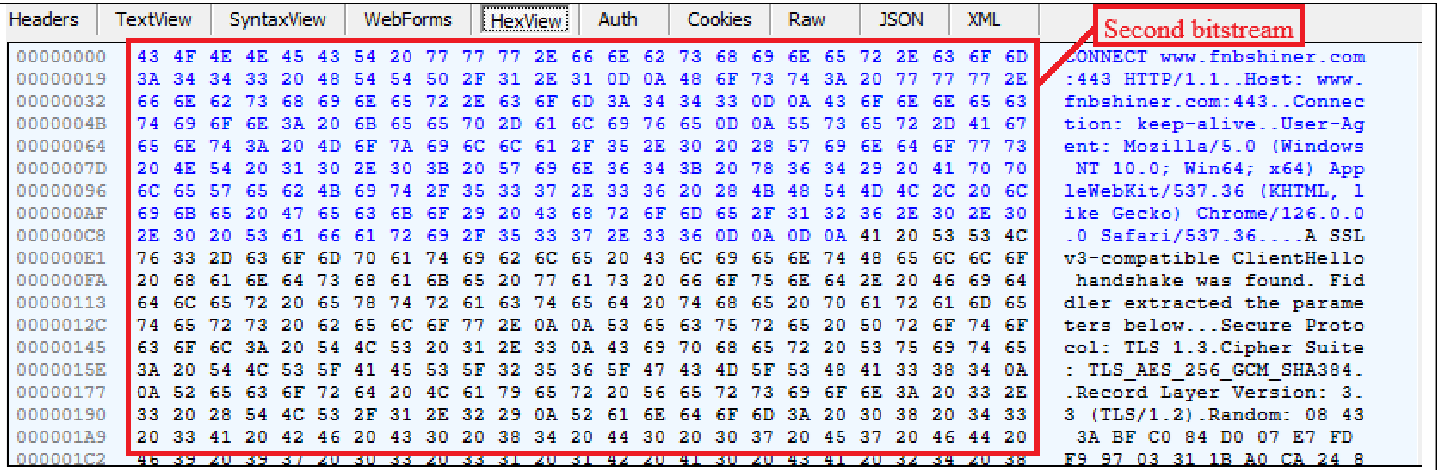
Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

https://datatracker.ietf.org/doc/html/rfc8446



Source: Fiddler Capture



Source: Fiddler Capture

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm.

As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message

	is communicated between the client and the server. **5. Record Protocol** The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients. TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see Appendix D.4). https://datatracker.ietf.org/doc/html/rfc8446#section-1 **2. Protocol Overview** Negotiating encryption algorithm The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic. https://datatracker.ietf.org/doc/html/rfc8446

	TLS consists of two primary components: - A handshake protocol (Section 4) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack. Negotiating encryption algos - A record protocol (Section 5) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys. https://datatracker.ietf.org/doc/html/rfc8446

	5.1. Record Layer The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^14 bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2. Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that: - Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **5.2. Record Payload Protection** The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see Section 5.3), and the additional data input is the record header. I.e., `additional_data = TLSCiphertext.opaque_type ‖` `TLSCiphertext.legacy_record_version ‖` `TLSCiphertext.length` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case. https://datatracker.ietf.org/doc/html/rfc5116

	2.1. Authenticated Encryption The authenticated encryption operation has four inputs, each of which is an octet string: A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom. A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in Section 3.2, and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces. A plaintext P, which contains the data to be encrypted and authenticated. The associated data A, which contains the data to be authenticated, but not encrypted. https://datatracker.ietf.org/doc/html/rfc5116

	The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1 Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [RFC5116], Section 4). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows: https://datatracker.ietf.org/doc/html/rfc8446#section-1

	This specification defines the following cipher suites for use with TLS 1.3. ``` +------------------------------+-------------+ | Description | Value | +------------------------------+-------------+ | TLS_AES_128_GCM_SHA256 | {0x13,0x01} | | | | | TLS_AES_256_GCM_SHA384 | {0x13,0x02} | | | | | TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} | | | | | TLS_AES_128_CCM_SHA256 | {0x13,0x04} | | | | | TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} | +------------------------------+-------------+ ``` https://datatracker.ietf.org/doc/html/rfc8446#section-1 All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4. Authentication Messages As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [sender]_handshake_traffic_secret. https://datatracker.ietf.org/doc/html/rfc8446#section-1

```
   Figure 1 below shows the basic full TLS handshake:

       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

Digital Content

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.1.1. Cryptographic Negotiation In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello: - A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports. Second encryption - A "supported_groups" (Section 4.2.7) extension which indicates the (EC)DHE groups which the client supports and a "key_share" (Section 4.2.8) extension which contains (EC)DHE shares for some or all of these groups. - A "signature_algorithms" (Section 4.2.3) extension which indicates the signature algorithms which the client can accept. A "signature_algorithms_cert" extension (Section 4.2.3) may also be added to indicate certificate-specific signature algorithms. First encryption - A "pre_shared_key" (Section 4.2.11) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" (Section 4.2.9) extension which indicates the key exchange modes that may be used with PSKs. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2. Certificate This message conveys the endpoint's certificate chain to the peer. The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK). The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2.3. Client Certificate Selection The following rules apply to certificates sent by the client: - The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]). - If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs. - The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS. - If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.3. Certificate Verify This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message. Structure of this message: `struct {` `    SignatureScheme algorithm;` `    opaque signature<0..2^16-1>;` `} CertificateVerify;` First decryption algorithm information The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely: https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	4.3.2. Certificate Request A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions. Structure of this message: `struct {` `opaque certificate_request_context<0..2^8-1>;` `Extension extensions<2..2^16-1>;` `} CertificateRequest;` https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages. extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions. https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	- RSASSA-PSS signature schemes are defined in Section 4.2.3. - The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello. - The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates. https://datatracker.ietf.org/doc/html/rfc8446#section-1 If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message. In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages. https://datatracker.ietf.org/doc/html/rfc8446#section-1

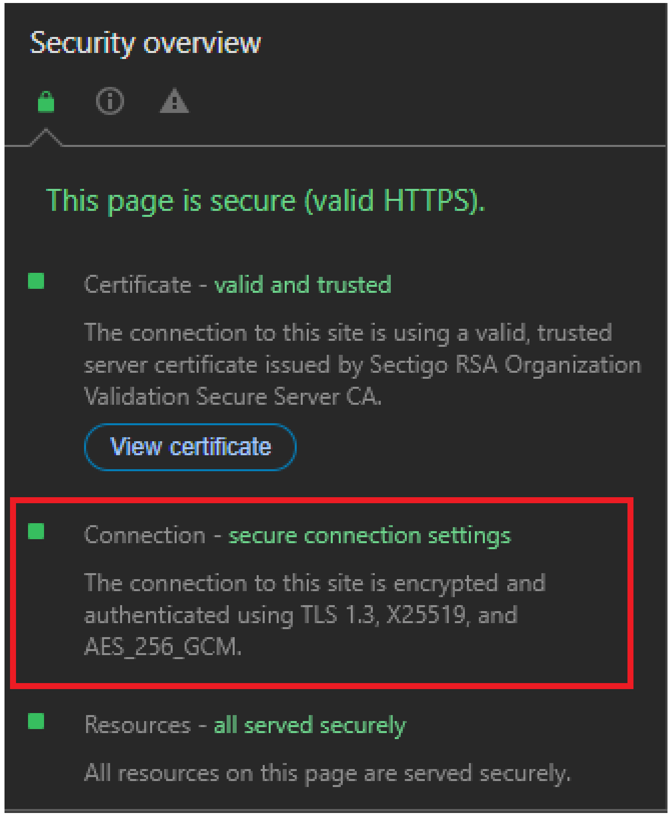
	4.2.3. Signature Algorithms TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2). The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension. https://datatracker.ietf.org/doc/html/rfc8446#section-1

<table>
<tr>
<td></td>
<td>

```
The "extension_data" field of these extensions contains a
SignatureSchemeList value:

   enum {
       /* RSASSA-PKCS1-v1_5 algorithms */
       rsa_pkcs1_sha256(0x0401),
       rsa_pkcs1_sha384(0x0501),
       rsa_pkcs1_sha512(0x0601),

       /* ECDSA algorithms */
       ecdsa_secp256r1_sha256(0x0403),
       ecdsa_secp384r1_sha384(0x0503),
       ecdsa_secp521r1_sha512(0x0603),

       /* RSASSA-PSS algorithms with public key OID rsaEncryption */
       rsa_pss_rsae_sha256(0x0804),
       rsa_pss_rsae_sha384(0x0805),
       rsa_pss_rsae_sha512(0x0806),

       /* EdDSA algorithms */
       ed25519(0x0807),
       ed448(0x0808),

       /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */
       rsa_pss_pss_sha256(0x0809),
       rsa_pss_pss_sha384(0x080a),
       rsa_pss_pss_sha512(0x080b),
```

https://datatracker.ietf.org/doc/html/rfc8446#section-1
</td>
</tr>
<tr>
<td>associating a second decryption algorithm with the second bit stream.</td>
<td>The standard practices associating a second decryption algorithm (e.g., cipher suit selected from one of the AEAD algorithms such as TLS_AES_256_GCM_SHA384, etc.) with the second bit stream (e.g., TLS ciphertext bitstream).

The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA algorithm) and generates a ciphertext.</td>
</tr>
</table>

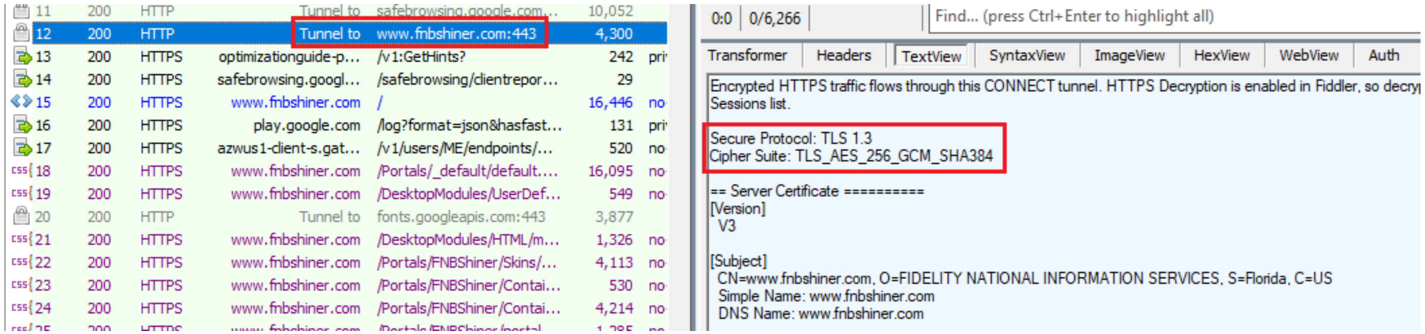
	The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it. The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate. The standard further practices encrypting the authentication message, including the encrypted digital certification and the certificate verify message, with a second decryption algorithm i.e., AEAD algorithm such as TLS_AES_256_GCM_SHA384, etc.  https://www.fnbshiner.com/

The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

https://datatracker.ietf.org/doc/html/rfc8446



Source: Fiddler Capture

Encrypted HTTPS traffic flows through this CONNECT tunnel. HTTPS Decryption is enabled in Fiddler, so decrypted sessions running in this tunnel will be shown in the Web Sessions list.

Secure Protocol: TLS 1.3
Cipher Suite: TLS_AES_256_GCM_SHA384 — Second decryption algorithm

== Server Certificate ==========
[Version]
V3

[Subject]
CN=www.fnbshiner.com, O=FIDELITY NATIONAL INFORMATION SERVICES, S=Florida, C=US
Simple Name: www.fnbshiner.com
DNS Name: www.fnbshiner.com

[Issuer]
CN=Sectigo RSA Organization Validation Secure Server CA, O=Sectigo Limited, L=Salford, S=Greater Manchester, C=GB
Simple Name: Sectigo RSA Organization Validation Secure Server CA
DNS Name: Sectigo RSA Organization Validation Secure Server CA

Source: Fiddler Capture

Headers | TextView | SyntaxView | WebForms | HexView | Auth | Cookies | Raw | JSON | XML

Second bitstream


```
00000000  43 4F 4E 4E 45 43 54 20 77 77 77 2E 66 6E 62 73 68 69 6E 65 72 2E 63 6F 6D  CONNECT www.fnbshiner.com
00000019  3A 34 34 33 20 48 54 54 50 2F 31 2E 31 0D 0A 48 6F 73 74 3A 20 77 77 77 2E  :443 HTTP/1.1..Host: www.
00000032  66 6E 62 73 68 69 6E 65 72 2E 63 6F 6D 3A 34 34 33 0D 0A 43 6F 6E 6E 65 63  fnbshiner.com:443..Connec
0000004B  74 69 6F 6E 3A 20 6B 65 65 70 2D 61 6C 69 76 65 0D 0A 55 73 65 72 2D 41 67  tion: keep-alive..User-Ag
00000064  65 6E 74 3A 20 4D 6F 7A 69 6C 6C 61 2F 35 2E 30 20 28 57 69 6E 64 6F 77 73  ent: Mozilla/5.0 (Windows
0000007D  20 4E 54 20 31 30 2E 30 3B 20 57 69 6E 36 34 3B 20 78 36 34 29 20 41 70 70   NT 10.0; Win64; x64) App
00000096  6C 65 57 65 62 4B 69 74 2F 35 33 37 2E 33 36 20 28 4B 48 54 4D 4C 2C 20 6C  leWebKit/537.36 (KHTML, l
000000AF  69 6B 65 20 47 65 63 6B 6F 29 20 43 68 72 6F 6D 65 2F 31 32 36 2E 30 2E 30  ike Gecko) Chrome/126.0.0
000000C8  2E 30 20 53 61 66 61 72 69 2F 35 33 37 2E 33 36 0D 0A 0D 0A 41 20 53 53 4C  .0 Safari/537.36....A SSL
000000E1  76 33 2D 63 6F 6D 70 61 74 69 62 6C 65 20 43 6C 69 65 6E 74 48 65 6C 6C 6F  v3-compatible ClientHello
000000FA  20 68 61 6E 64 73 68 61 6B 65 20 77 61 73 20 66 6F 75 6E 64 2E 20 46 69 64   handshake was found. Fid
00000113  64 6C 65 72 20 65 78 74 72 61 63 74 65 64 20 74 68 65 20 70 61 72 61 6D 65  dler extracted the parame
0000012C  74 65 72 73 20 62 65 6C 6F 77 2E 0A 0A 53 65 63 75 72 65 20 50 72 6F 74 6F  ters below...Secure Proto
00000145  63 6F 6C 3A 20 54 4C 53 20 31 2E 33 0A 43 69 70 68 65 72 20 53 75 69 74 65  col: TLS 1.3.Cipher Suite
0000015E  3A 20 54 4C 53 5F 41 45 53 5F 32 35 36 5F 47 43 4D 5F 53 48 41 33 38 34 0A  : TLS_AES_256_GCM_SHA384.
00000177  0A 52 65 63 6F 72 64 20 4C 61 79 65 72 20 56 65 72 73 69 6F 6E 3A 20 33 2E  .Record Layer Version: 3.
00000190  33 20 28 54 4C 53 2F 31 2E 32 29 0A 52 61 6E 64 6F 6D 3A 20 30 38 20 34 33  3 (TLS/1.2).Random: 08 43
000001A9  20 33 41 20 42 46 20 43 30 20 38 34 20 44 30 20 30 37 20 45 37 20 46 44 20   3A BF C0 84 D0 07 E7 FD 
000001C2  46 39 20 39 37 20 30 33 20 33 31 20 31 42 20 41 30 20 43 41 20 32 34 20 38  F9 97 03 31 1B A0 CA 24 8
```


Source: Fiddler Capture

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for

	encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm. As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server. Further, the AEAD encrypted message comprises a ciphertext (e.g., encrypted ciphertext after the encryption by the second encryption algorithm), nonce (e.g., associating second decryption algo), key and associated data. The maximum length of nonce is a cipher suit specific element. The nonce and associated data are utilized in decryption of the AEAD encrypted message.

	5. Record Protocol The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients. TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see Appendix D.4). https://datatracker.ietf.org/doc/html/rfc8446#section-1 **2. Protocol Overview** Negotiating encryption algorithm The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic. https://datatracker.ietf.org/doc/html/rfc8446

	TLS consists of two primary components: - A handshake protocol (Section 4) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack. Negotiating encryption algos - A record protocol (Section 5) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys. https://datatracker.ietf.org/doc/html/rfc8446

	5.1. Record Layer The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^14 bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2. Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that: - Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **5.2. Record Payload Protection** The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see Section 5.3), and the additional data input is the record header. I.e., `additional_data = TLSCiphertext.opaque_type || TLSCiphertext.legacy_record_version || TLSCiphertext.length` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case. https://datatracker.ietf.org/doc/html/rfc5116

	2.1. Authenticated Encryption The authenticated encryption operation has four inputs, each of which is an octet string: A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom. A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in Section 3.2, and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces. A plaintext P, which contains the data to be encrypted and authenticated. The associated data A, which contains the data to be authenticated, but not encrypted. https://datatracker.ietf.org/doc/html/rfc5116

	2.2. Authenticated Decryption Second decryption algorithm The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure). https://datatracker.ietf.org/doc/html/rfc5116 The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [RFC5116], Section 4). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows: https://datatracker.ietf.org/doc/html/rfc8446#section-1 This specification defines the following cipher suites for use with TLS 1.3. ``` +------------------------------+-------------+ | Description | Value | +------------------------------+-------------+ | TLS_AES_128_GCM_SHA256 | {0x13,0x01} | | | | | TLS_AES_256_GCM_SHA384 | {0x13,0x02} | | | | | TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} | | | | | TLS_AES_128_CCM_SHA256 | {0x13,0x04} | | | | | TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} | +------------------------------+-------------+ ``` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **4.4. Authentication Messages** As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [sender]_handshake_traffic_secret. https://datatracker.ietf.org/doc/html/rfc8446#section-1

Figure 1 below shows the basic full TLS handshake:

```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

Digital Content

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2. Certificate This message conveys the endpoint's certificate chain to the peer. The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK). The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2.3. Client Certificate Selection The following rules apply to certificates sent by the client: - The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]). - If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs. - The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS. - If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.3. Certificate Verify This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message. Structure of this message: `struct {` `   SignatureScheme algorithm;` `   opaque signature<0..2^16-1>;` `} CertificateVerify;` The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely: https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

<table><tr><td></td><td>

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {
    opaque certificate_request_context<0..2^8-1>;
    Extension extensions<2..2^16-1>;
} CertificateRequest;
```

https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

</td></tr></table>

	certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages. extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions. https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	- RSASSA-PSS signature schemes are defined in Section 4.2.3. - The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello. - The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates. https://datatracker.ietf.org/doc/html/rfc8446#section-1 If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message. In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.2.3. Signature Algorithms TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2). The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension. https://datatracker.ietf.org/doc/html/rfc8446#section-1

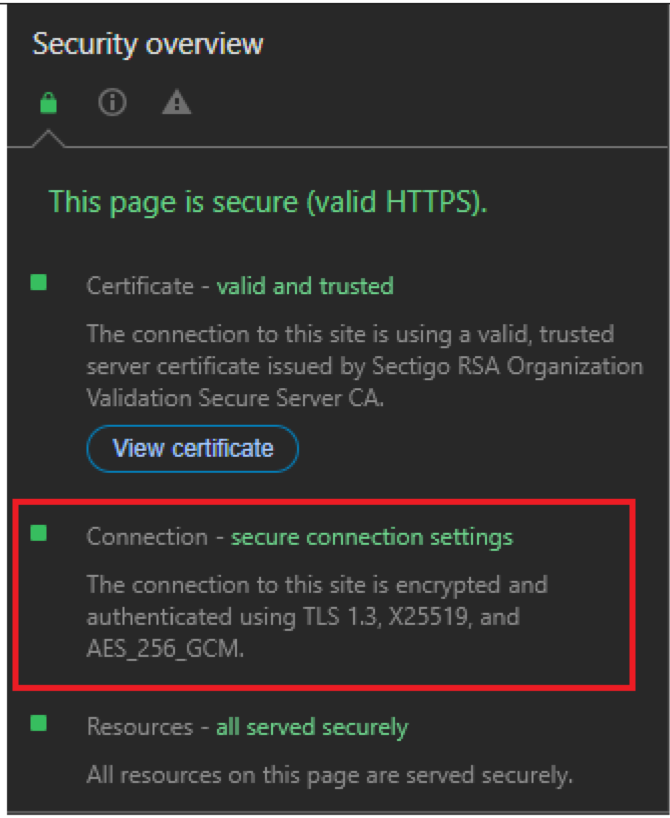
<table>
<tr>
<td></td>
<td>

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
   enum {
       /* RSASSA-PKCS1-v1_5 algorithms */
       rsa_pkcs1_sha256(0x0401),
       rsa_pkcs1_sha384(0x0501),
       rsa_pkcs1_sha512(0x0601),

       /* ECDSA algorithms */
       ecdsa_secp256r1_sha256(0x0403),
       ecdsa_secp384r1_sha384(0x0503),
       ecdsa_secp521r1_sha512(0x0603),

       /* RSASSA-PSS algorithms with public key OID rsaEncryption */
       rsa_pss_rsae_sha256(0x0804),
       rsa_pss_rsae_sha384(0x0805),
       rsa_pss_rsae_sha512(0x0806),

       /* EdDSA algorithms */
       ed25519(0x0807),
       ed448(0x0808),

       /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */
       rsa_pss_pss_sha256(0x0809),
       rsa_pss_pss_sha384(0x080a),
       rsa_pss_pss_sha512(0x080b),
```

https://datatracker.ietf.org/doc/html/rfc8446#section-1
</td>
</tr>
<tr>
<td>2. The method of claim 1, further comprising decrypting the first bit stream and the second</td>
<td>The standard further discloses decrypting the first bit stream (e.g., encrypted digital certificate with signature encryption algorithm i.e., SHA-256 RSA, etc.) and the second bit stream (e.g., a second-level encryption with AEAD encryption algorithm such as TLS_AES_256_GCM_SHA384, etc.) with the first associated decryption</td>
</tr>
</table>

bit stream with the first associated decryption algorithm and the second associated decryption algorithm wherein the decryption is accomplished by a target unit.	algorithm (e.g., signature decryption algorithm i.e., SHA-256 RSA, etc.) and the second associated decryption algorithm (e.g., cipher suit selected from one of the AEAD decryption algorithms such as TLS_AES_256_GCM_SHA384, etc.) wherein the decryption is accomplished by a target unit (e.g., a server of the accused instrumentality). The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA algorithm) and generates a ciphertext. The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it. The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate. The standard further practices encrypting the authentication message, including the encrypted digital certification and the certificate verify message, with a second decryption algorithm i.e., AEAD algorithm such as TLS_AES_256_GCM_SHA384, etc.

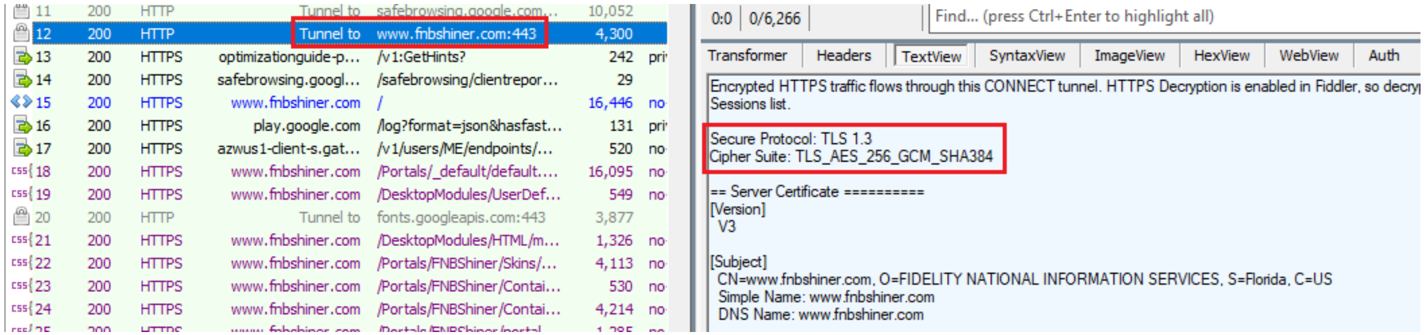
	 https://www.fnbshiner.com/

The Transport Layer Security (TLS) Protocol Version 1.3

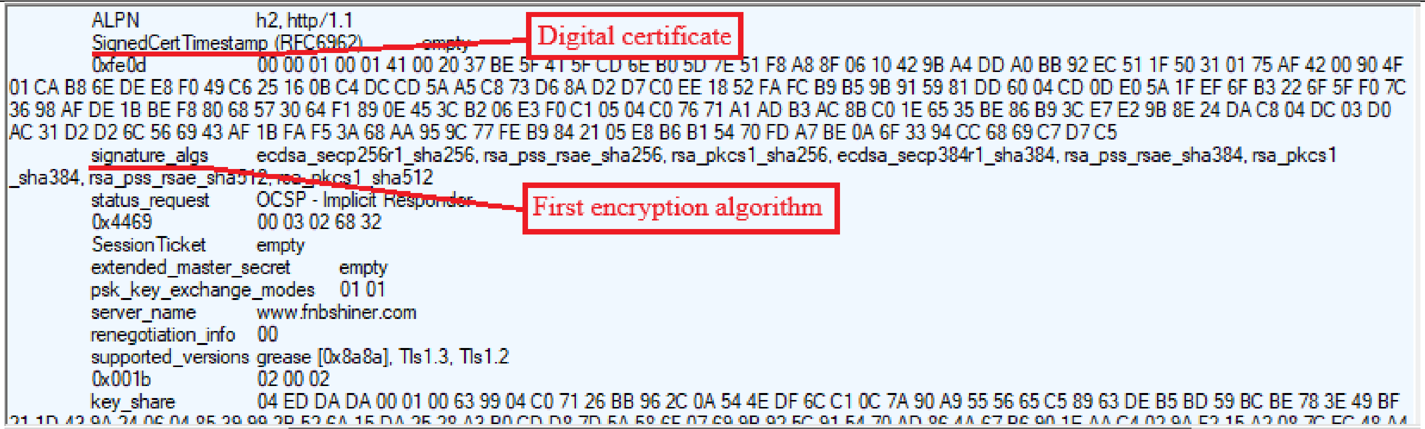
Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

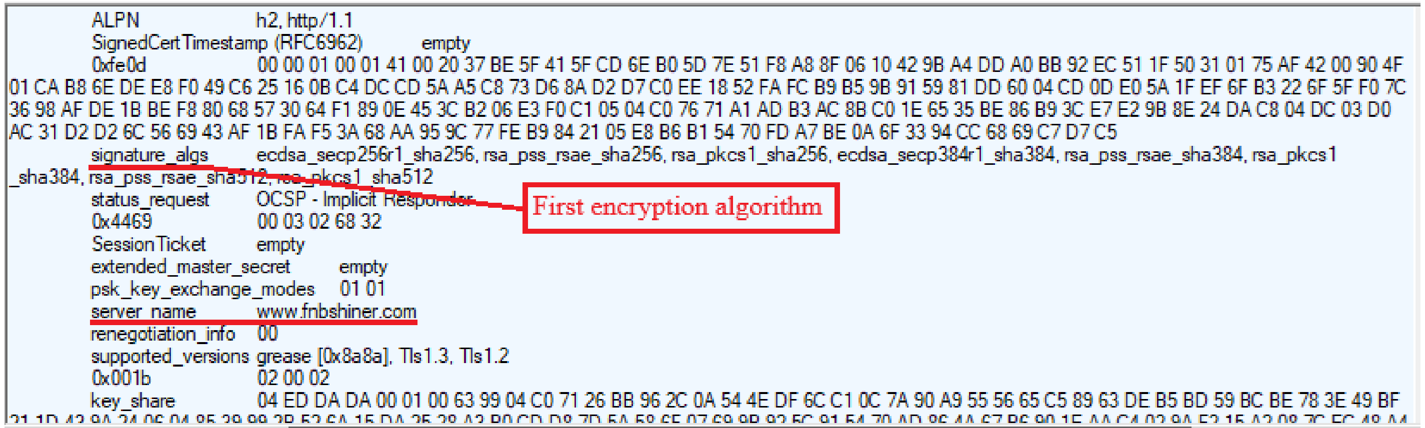
https://datatracker.ietf.org/doc/html/rfc8446



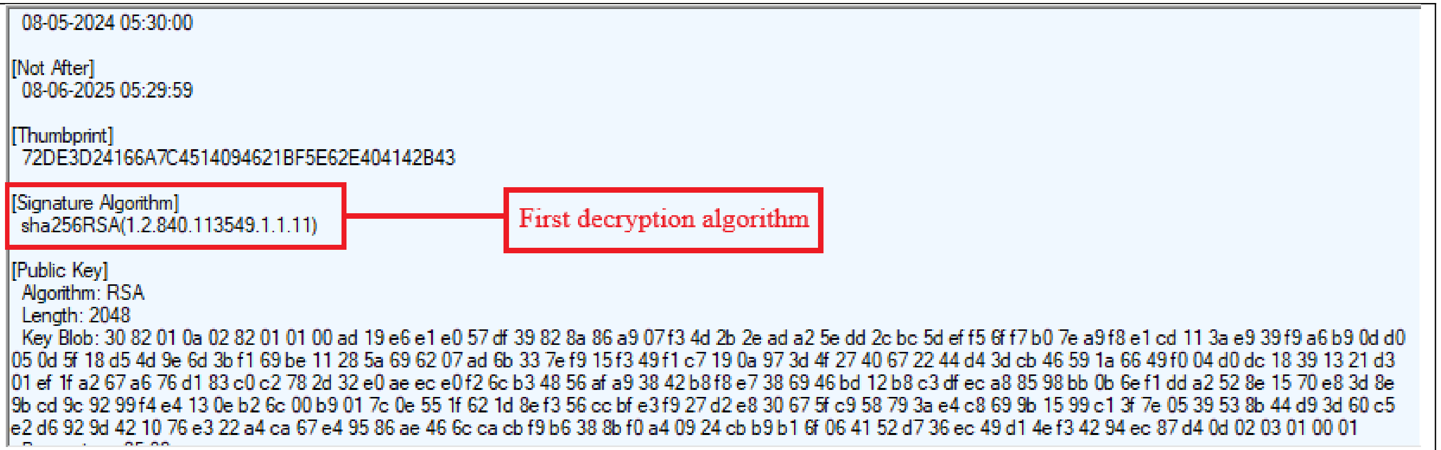
Source: Fiddler Capture



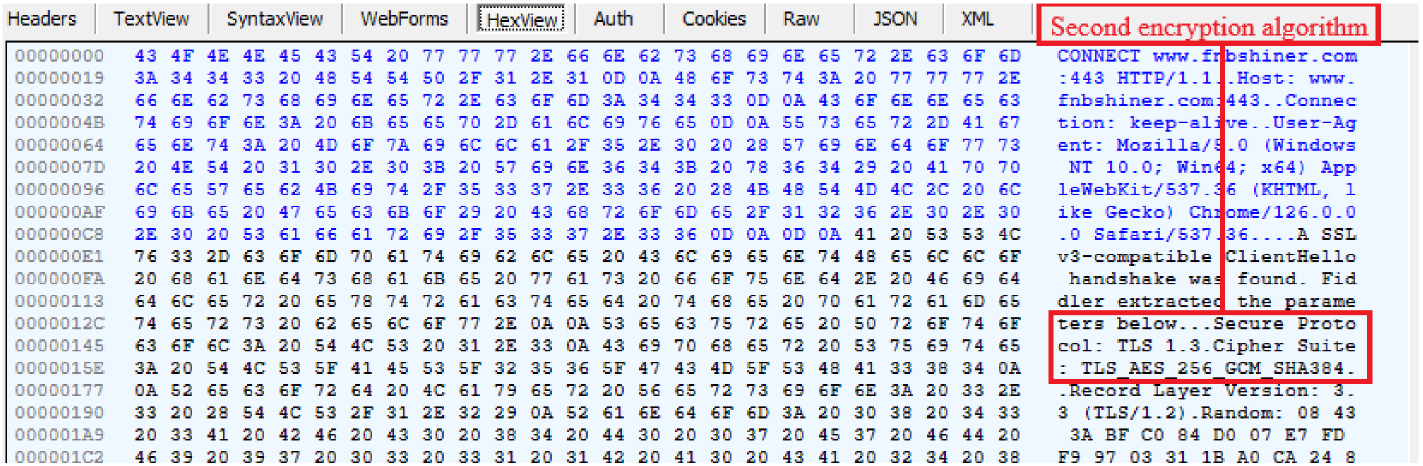
Source: Fiddler Capture



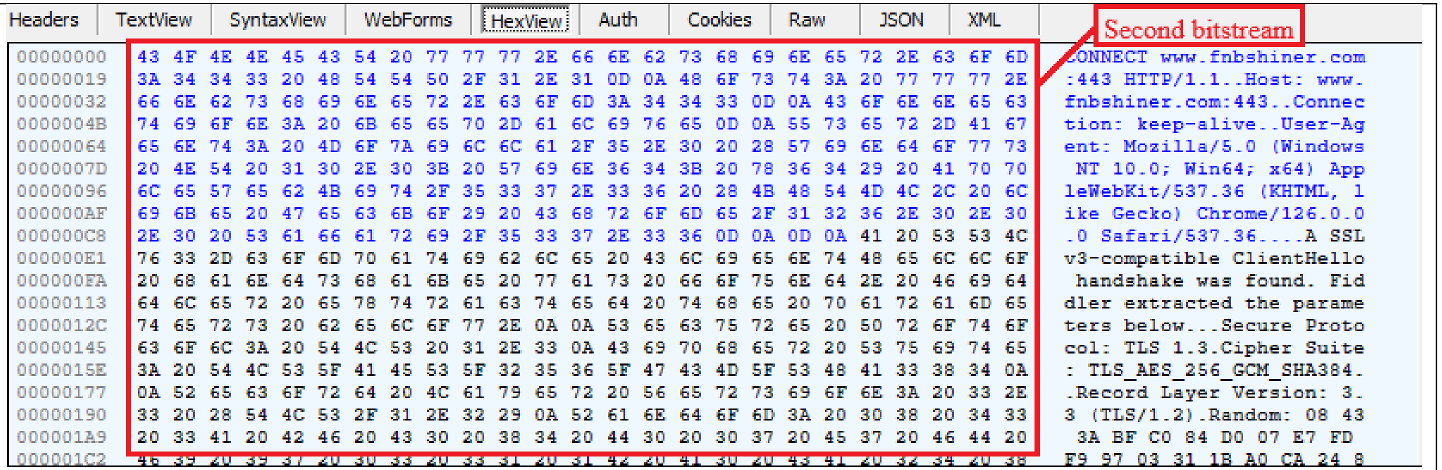
Source: Fiddler Capture



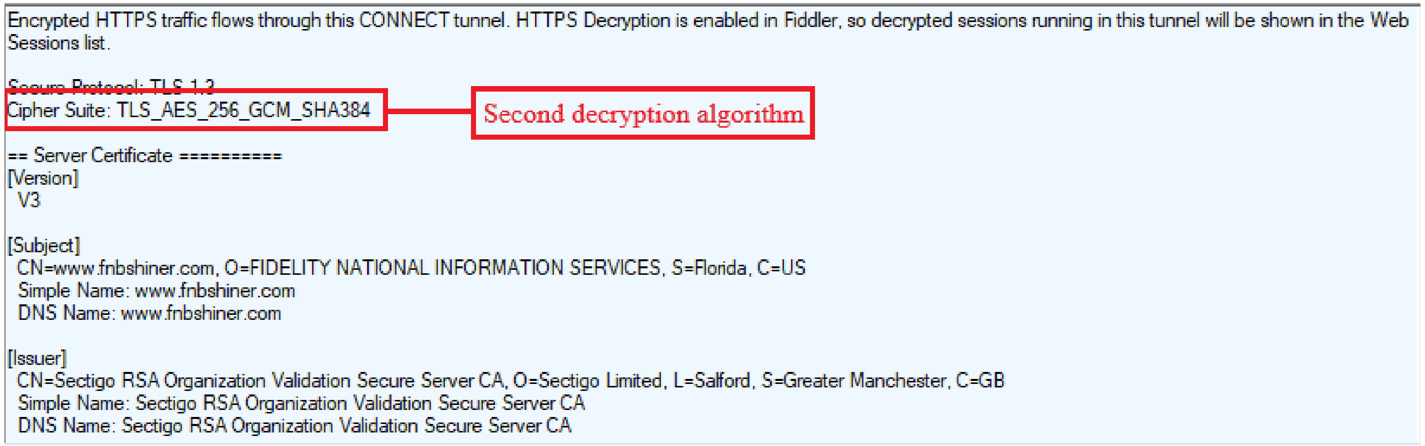
Source: Fiddler Capture



Source: Fiddler Capture

Source: Fiddler Capture



Source: Fiddler Capture

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for

	encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm. As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server. Further, the AEAD encrypted message comprises a ciphertext (e.g., encrypted ciphertext after the encryption by the second encryption algorithm), nonce (e.g., associating second decryption algo), key and associated data. The maximum length of nonce is a cipher suit specific element. The nonce and associated data are utilized in decryption of the AEAD encrypted message.

	5. Record Protocol The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients. TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see Appendix D.4). https://datatracker.ietf.org/doc/html/rfc8446#section-1 **2. Protocol Overview** Negotiating encryption algorithm The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic. https://datatracker.ietf.org/doc/html/rfc8446

	TLS consists of two primary components: - A handshake protocol (Section 4) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack. Negotiating encryption algos - A record protocol (Section 5) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys. https://datatracker.ietf.org/doc/html/rfc8446

	5.1. Record Layer The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^14 bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2. Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that: - Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **5.2. Record Payload Protection** The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see Section 5.3), and the additional data input is the record header. I.e., `additional_data = TLSCiphertext.opaque_type ||` `TLSCiphertext.legacy_record_version ||` `TLSCiphertext.length` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case. https://datatracker.ietf.org/doc/html/rfc5116

	2.1. Authenticated Encryption The authenticated encryption operation has four inputs, each of which is an octet string: A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom. A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in Section 3.2, and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces. A plaintext P, which contains the data to be encrypted and authenticated. The associated data A, which contains the data to be authenticated, but not encrypted. https://datatracker.ietf.org/doc/html/rfc5116

	2.2. Authenticated Decryption Second decryption algorithm The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure). https://datatracker.ietf.org/doc/html/rfc5116 The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [RFC5116], Section 4). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows: https://datatracker.ietf.org/doc/html/rfc8446#section-1 This specification defines the following cipher suites for use with TLS 1.3. Description | Value TLS_AES_128_GCM_SHA256 | {0x13,0x01} TLS_AES_256_GCM_SHA384 | {0x13,0x02} TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} TLS_AES_128_CCM_SHA256 | {0x13,0x04} TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} https://datatracker.ietf.org/doc/html/rfc8446#section-1

	All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **4.4. Authentication Messages** As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [sender]_handshake_traffic_secret. https://datatracker.ietf.org/doc/html/rfc8446#section-1

```
   Figure 1 below shows the basic full TLS handshake:

       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

Digital Content

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2. Certificate This message conveys the endpoint's certificate chain to the peer. The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK). The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2.3. Client Certificate Selection The following rules apply to certificates sent by the client: - The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]). - If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs. - The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS. - If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.3. Certificate Verify This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message. Structure of this message: `struct {` `SignatureScheme algorithm;` `opaque signature<0..2^16-1>;` `} CertificateVerify;` The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely: https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	4.3.2. Certificate Request A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions. Structure of this message: `struct {` `opaque certificate_request_context<0..2^8-1>;` `Extension extensions<2..2^16-1>;` `} CertificateRequest;` https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages. extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions. https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

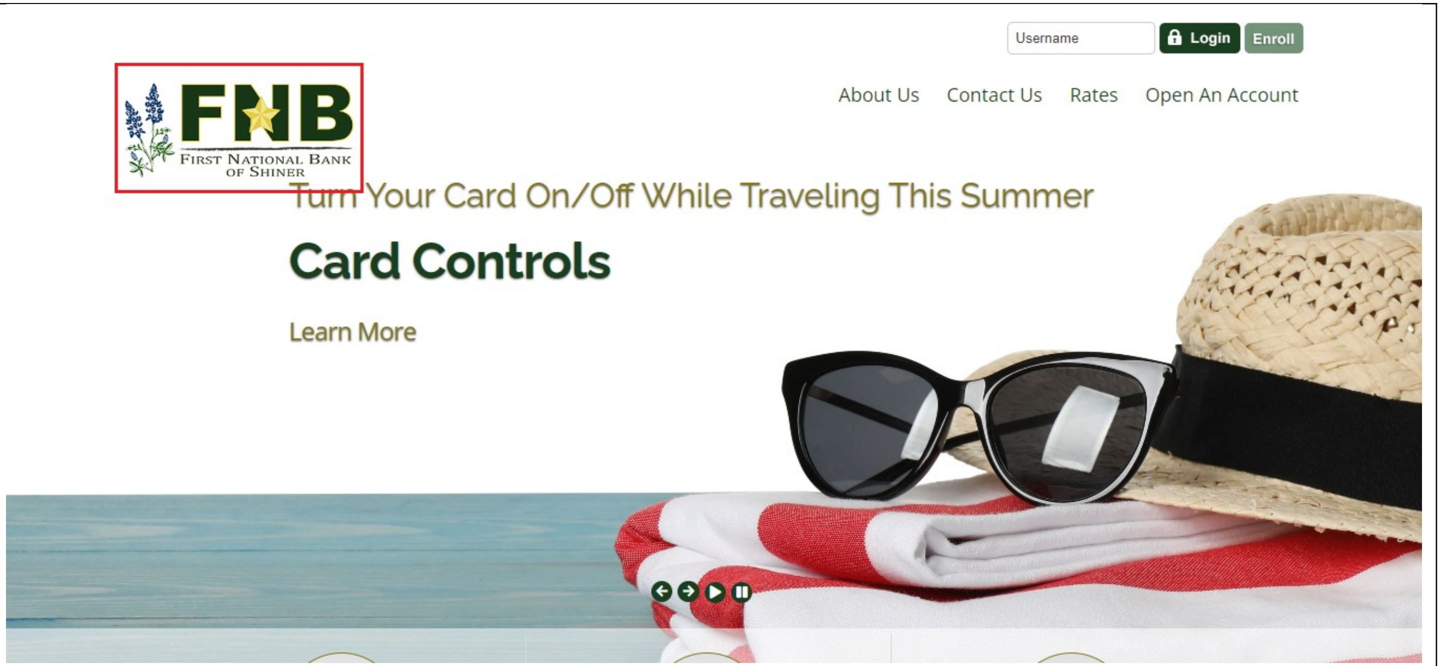
	- RSASSA-PSS signature schemes are defined in Section 4.2.3. - The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello. - The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates. https://datatracker.ietf.org/doc/html/rfc8446#section-1 If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message. In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages. https://datatracker.ietf.org/doc/html/rfc8446#section-1

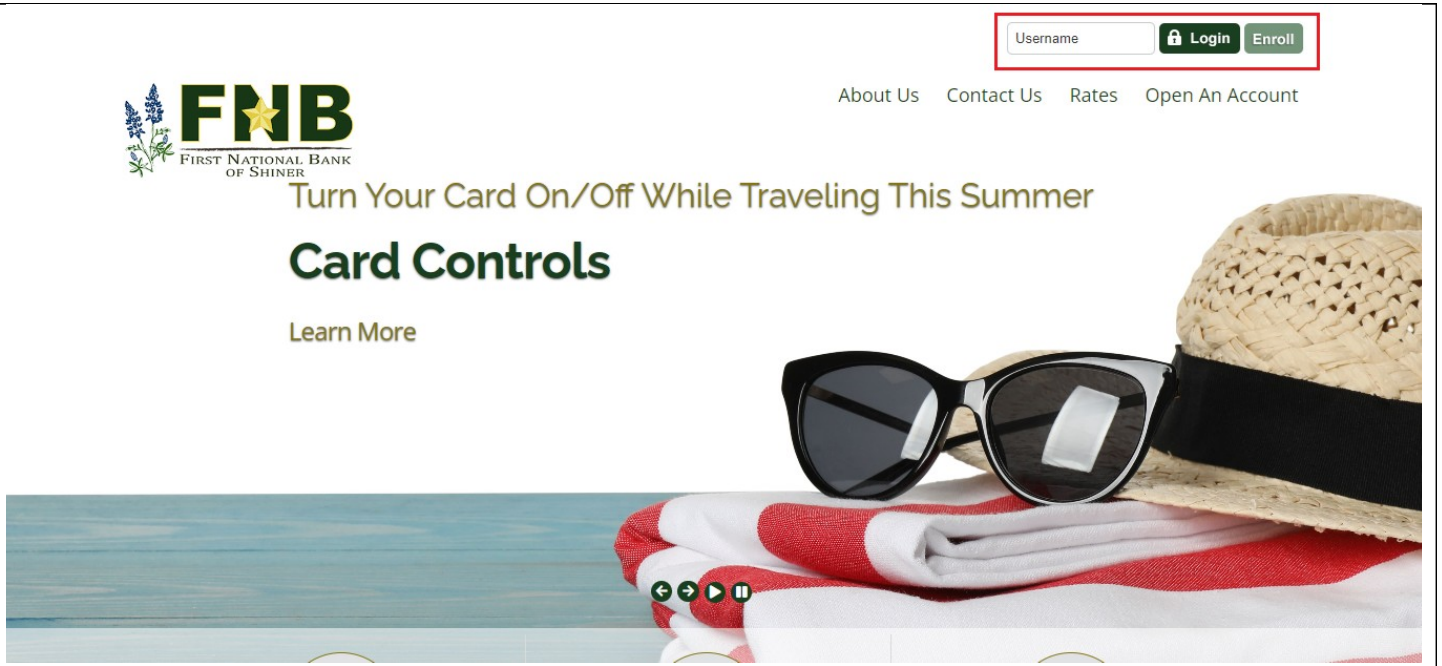
	4.2.3. Signature Algorithms TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2). The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension. https://datatracker.ietf.org/doc/html/rfc8446#section-1

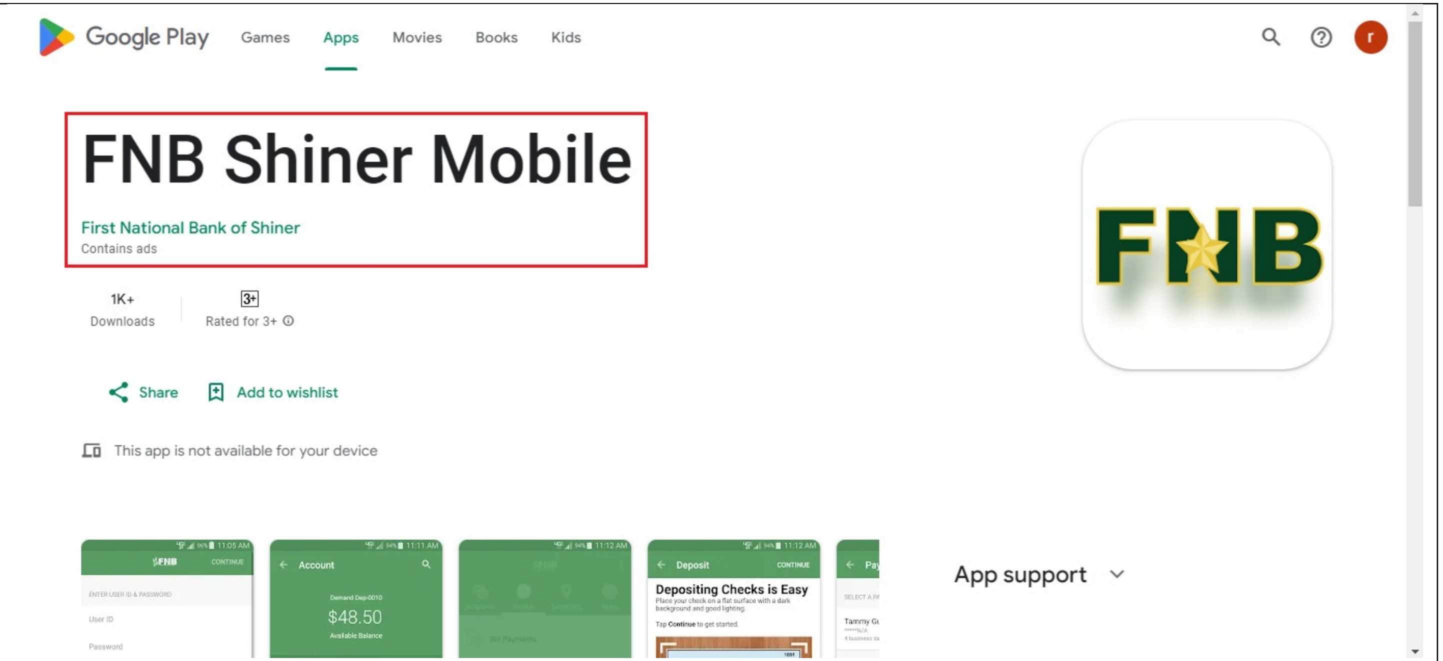
	The "extension_data" field of these extensions contains a SignatureSchemeList value: `enum {` `    /* RSASSA-PKCS1-v1_5 algorithms */` `    rsa_pkcs1_sha256(0x0401),` `    rsa_pkcs1_sha384(0x0501),` `    rsa_pkcs1_sha512(0x0601),` `    /* ECDSA algorithms */` `    ecdsa_secp256r1_sha256(0x0403),` `    ecdsa_secp384r1_sha384(0x0503),` `    ecdsa_secp521r1_sha512(0x0603),` `    /* RSASSA-PSS algorithms with public key OID rsaEncryption */` `    rsa_pss_rsae_sha256(0x0804),` `    rsa_pss_rsae_sha384(0x0805),` `    rsa_pss_rsae_sha512(0x0806),` `    /* EdDSA algorithms */` `    ed25519(0x0807),` `    ed448(0x0808),` `    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */` `    rsa_pss_pss_sha256(0x0809),` `    rsa_pss_pss_sha384(0x080a),` `    rsa_pss_pss_sha512(0x080b),` https://datatracker.ietf.org/doc/html/rfc8446#section-1 As shown below, the receiving party will be able to decrypt the encrypted message with the provided signature decryption algorithm information i.e., SHA-256 RSA decryption algorithm.

	We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q, compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de = 1 \pmod{\varphi(n)}$. we know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$. Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is $\boxed{c = m^e \bmod n.}$ First encryption To decrypt, we compute $c^d \bmod n$ to obtain $$c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$$ The result of Exercise 3.13 tells us that $\boxed{m \equiv m^{1+k\varphi(n)} \pmod{n},}$ First decryption https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

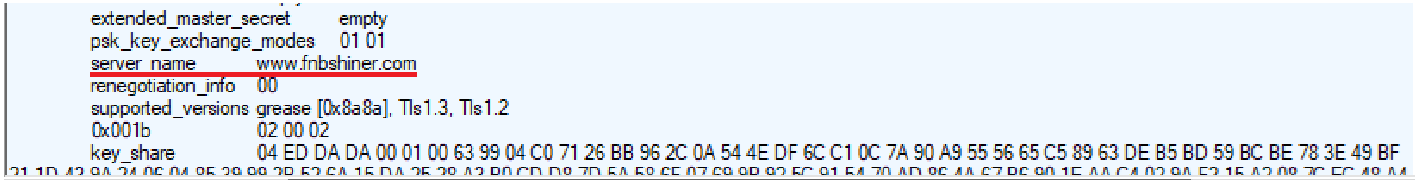
	The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf
3. The method of claim 2, wherein the decrypting is done using a key associated with each decryption algorithm.	The standard practices the method such that the decrypting is done using a key (e.g., decryption key) associated with each decryption algorithm (e.g., signature decryption algorithm such as SHA-256RSA, etc., and AEAD decryption algorithm such as TLS_AES_256_GCM_SHA384, etc.).

	 https://www.fnbshiner.com/

	 https://www.fnbshiner.com/



https://play.google.com/store/apps/details?id=com.mfoundry.mb.android.mb_113106833&hl=en_US



extended_master_secret	empty
psk_key_exchange_modes	01 01
server_name	www.fnbshiner.com
renegotiation_info	00
supported_versions	grease [0x8a8a], Tls1.3, Tls1.2
0x001b	02 00 02
key_share	04 ED DA DA 00 01 00 63 99 04 C0 71 26 BB 96 2C 0A 54 4E DF 6C C1 0C 7A 90 A9 55 56 65 C5 89 63 DE B5 BD 59 BC BE 78 3E 49 BF

Source: Fiddler Capture

As shown below, the signature decryption algorithm utilizes a private key for a first decryption and the AEAD decryption algorithm uses a key K. Both the decryption techniques are decrypting using their respective associated keys.

	The "extension_data" field of these extensions contains a SignatureSchemeList value: `enum {` `    /* RSASSA-PKCS1-v1_5 algorithms */` `    rsa_pkcs1_sha256(0x0401),` `    rsa_pkcs1_sha384(0x0501),` `    rsa_pkcs1_sha512(0x0601),` `    /* ECDSA algorithms */` `    ecdsa_secp256r1_sha256(0x0403),` `    ecdsa_secp384r1_sha384(0x0503),` `    ecdsa_secp521r1_sha512(0x0603),` `    /* RSASSA-PSS algorithms with public key OID rsaEncryption */` `    rsa_pss_rsae_sha256(0x0804),` `    rsa_pss_rsae_sha384(0x0805),` `    rsa_pss_rsae_sha512(0x0806),` `    /* EdDSA algorithms */` `    ed25519(0x0807),` `    ed448(0x0808),` `    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */` `    rsa_pss_pss_sha256(0x0809),` `    rsa_pss_pss_sha384(0x080a),` `    rsa_pss_pss_sha512(0x080b),` https://datatracker.ietf.org/doc/html/rfc8446#section-1

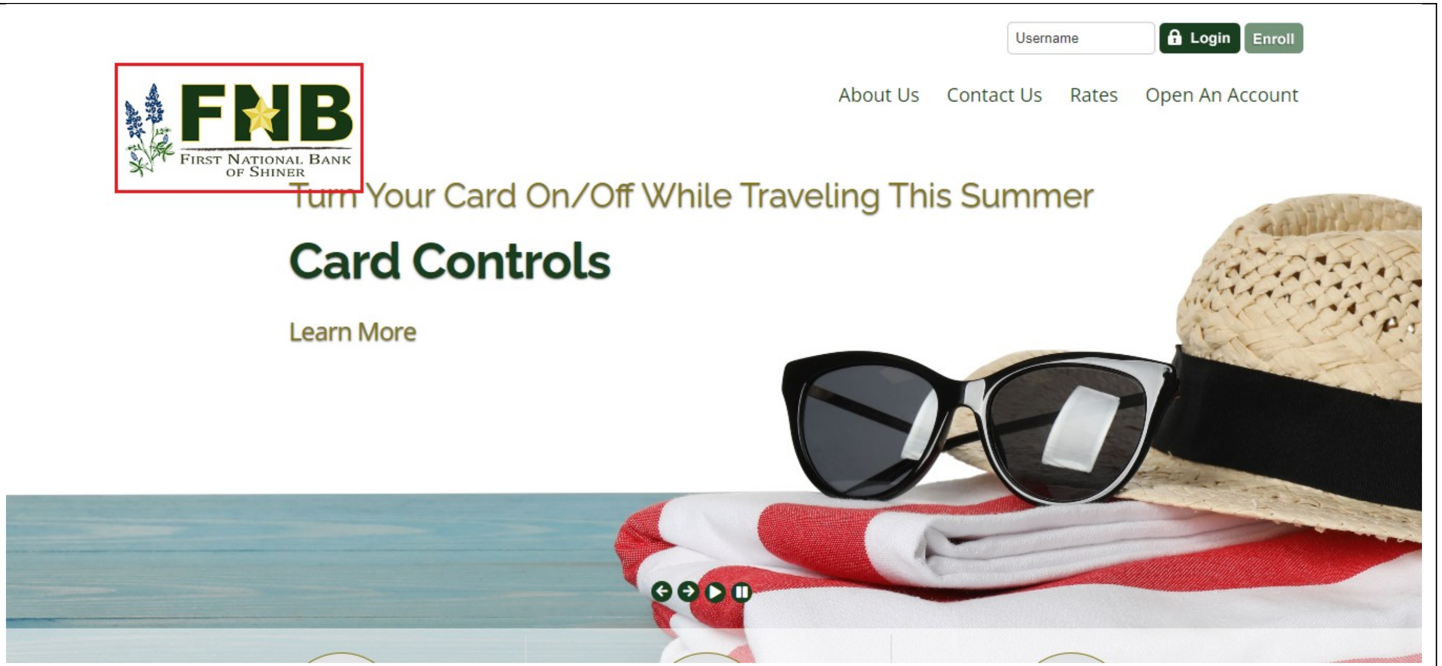
	There is also a decryption function D that takes a ciphertext and a decryption key K_D, and reproduces the plaintext message. $D(C, K_D) = P$ In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster. In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B's encryption key, encrypt a message with it, and send the result to person B. Only someone with B's decryption key, namely only B, can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

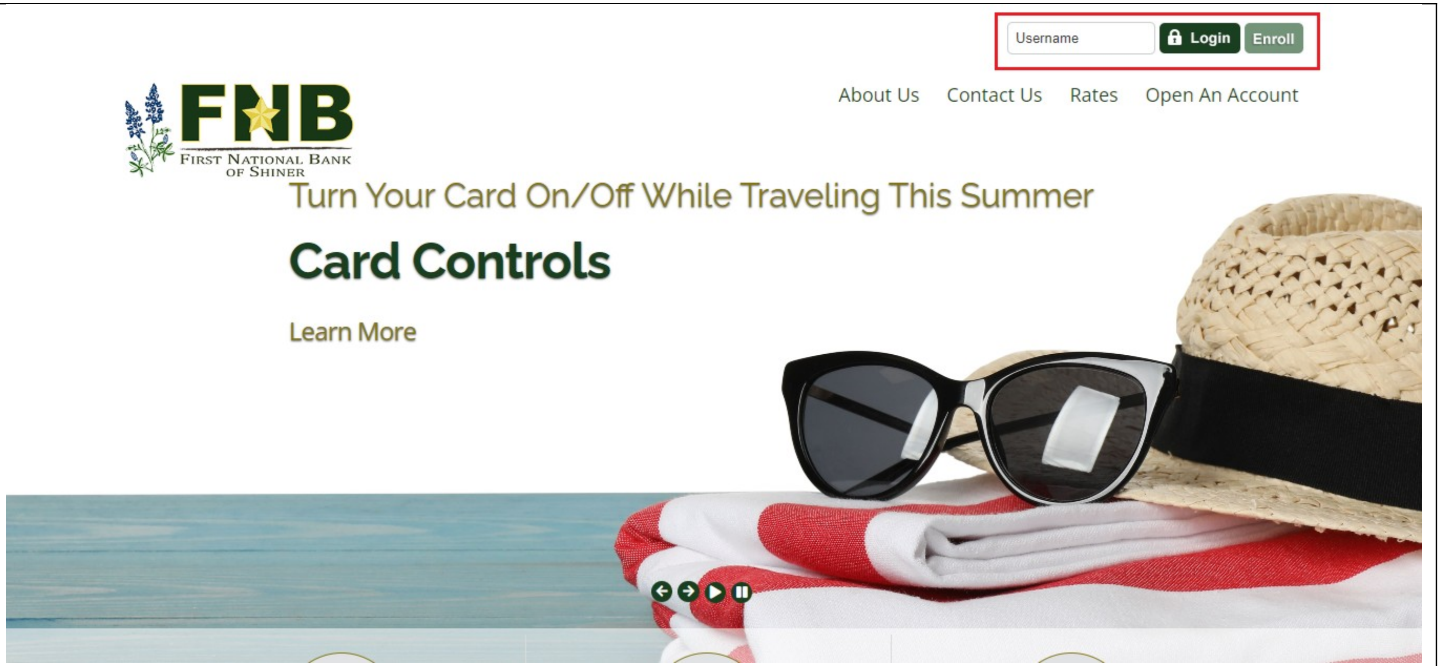
	We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q, compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de = 1 \pmod{\varphi(n)}$. we know that $de-1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$. Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is $\boxed{c = m^e \bmod n.}$ First encryption To decrypt, we compute $c^d \bmod n$ to obtain $$c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$$ The result of Exercise 3.13 tells us that $\boxed{m \equiv m^{1+k\varphi(n)} \pmod{n},}$ First decryption https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions `MD5`, `SHA-1`, and `SHA-256`. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

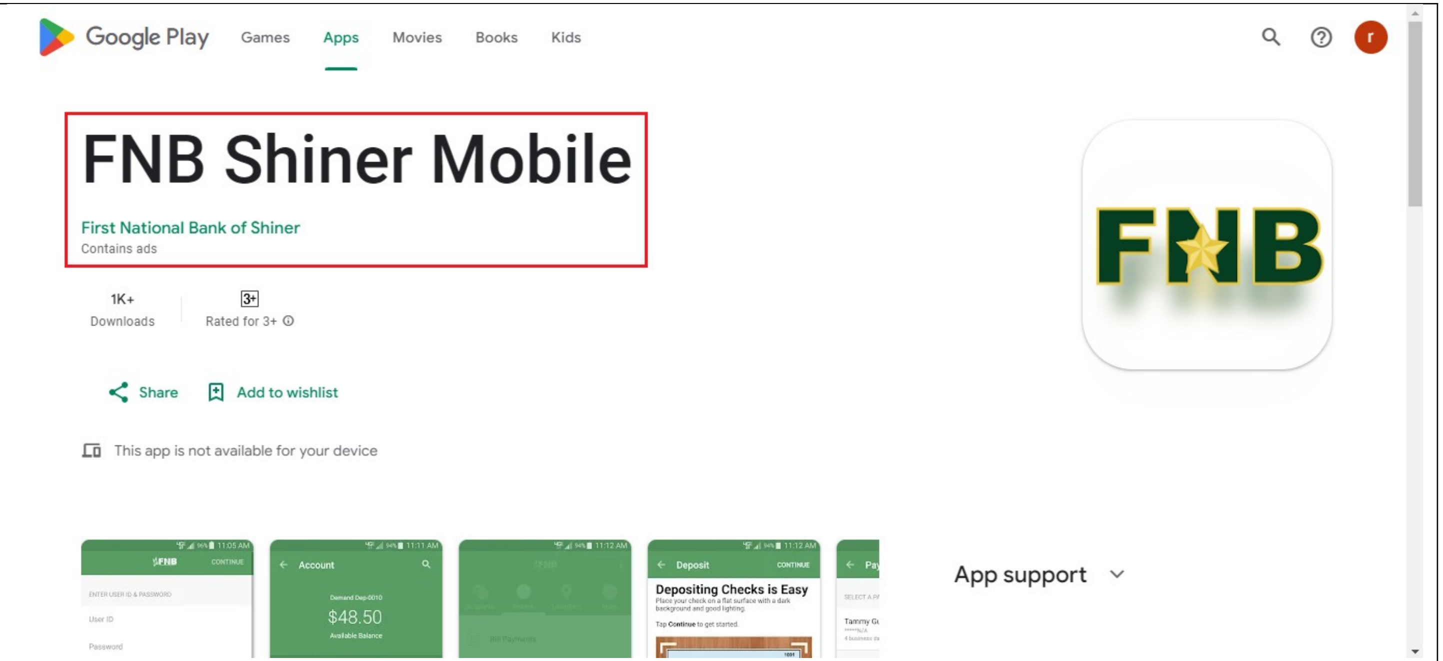
	This specification defines the following cipher suites for use with TLS 1.3. | Description | Value | | TLS_AES_128_GCM_SHA256 | {0x13,0x01} | | TLS_AES_256_GCM_SHA384 | {0x13,0x02} | | TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} | | TLS_AES_128_CCM_SHA256 | {0x13,0x04} | | TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} | https://datatracker.ietf.org/doc/html/rfc8446#section-1

	2.2. Authenticated Decryption The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure). https://datatracker.ietf.org/doc/html/rfc5116 The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1
4. The method of claim 3, wherein the key is resident in hardware of the target unit or the key is retrieved from a server.	The standard utilized by the accused instrumentality practices the method such that the key is resident in hardware (e.g., stored in a memory storage of the server such as a database, RAM, etc.) of the target unit (e.g., server of the accused instrumentality) or the key is retrieved from a server.

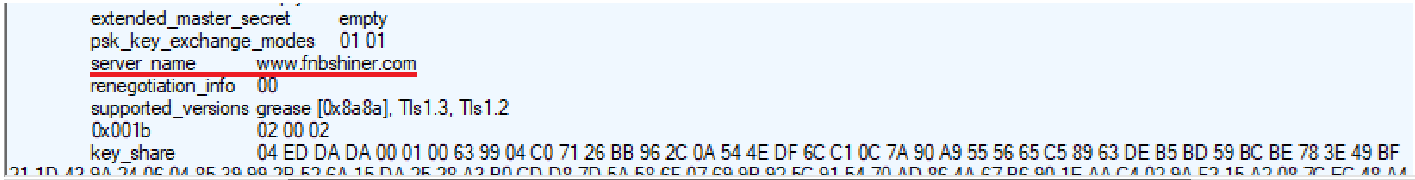
	 https://www.fnbshiner.com/

<table>
<tr>
<td></td>
<td>


https://www.fnbshiner.com/
</td>
</tr>
</table>



https://play.google.com/store/apps/details?id=com.mfoundry.mb.android.mb_113106833&hl=en_US



Source: Fiddler Capture

Tech Accelerator

Server hardware guide: Architecture, products and management

3. Random access memory

RAM is the main type of memory in a computing system. RAM holds the software instructions and data needed by the processor, along with any output from the processor, such as data to be moved to a storage device. Thus, RAM works very closely with the processor and must match the processor's incredible speed and performance. This kind of fast memory is usually termed dynamic RAM, and several DRAM variations are available for servers.

https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms

Tech Accelerator

Server hardware guide: Architecture, products and management

4. Hard disk drive

This hardware is responsible for reading, writing and positioning of the hard disk, which is one technology for data storage on server hardware. Developed at IBM in 1953, the [hard disk drive (HDD)]() has evolved over time from the size of a refrigerator to the standard 2.5-inch and 3.5-inch form factors.

https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms

As shown below, the server comprises a memory storage to store messages for establishing secure TLS communication. the standard discloses multiple signature encryption algorithms for a first encryption and multiple AEAD encryption algorithms for the second encryption. A signature decryption algorithm utilizes a private key for decrypting the first bitstream encrypted with the signature encryption and an AEAD decryption algorithm uses a key K for decrypting the second bitstream encrypted with the AEAD encryption. Both the decryption techniques are decrypting using their respective associated keys. A server must have a storage to store information pertaining to these algorithms and their corresponding keys such as private key, Key K, etc., to establish secure TLS communication with a client.

	Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT. https://datatracker.ietf.org/doc/html/rfc8446#

	The "extension_data" field of these extensions contains a SignatureSchemeList value: `enum {` `    /* RSASSA-PKCS1-v1_5 algorithms */` `    rsa_pkcs1_sha256(0x0401),` `    rsa_pkcs1_sha384(0x0501),` `    rsa_pkcs1_sha512(0x0601),` `    /* ECDSA algorithms */` `    ecdsa_secp256r1_sha256(0x0403),` `    ecdsa_secp384r1_sha384(0x0503),` `    ecdsa_secp521r1_sha512(0x0603),` `    /* RSASSA-PSS algorithms with public key OID rsaEncryption */` `    rsa_pss_rsae_sha256(0x0804),` `    rsa_pss_rsae_sha384(0x0805),` `    rsa_pss_rsae_sha512(0x0806),` `    /* EdDSA algorithms */` `    ed25519(0x0807),` `    ed448(0x0808),` `    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */` `    rsa_pss_pss_sha256(0x0809),` `    rsa_pss_pss_sha384(0x080a),` `    rsa_pss_pss_sha512(0x080b),` https://datatracker.ietf.org/doc/html/rfc8446#section-1

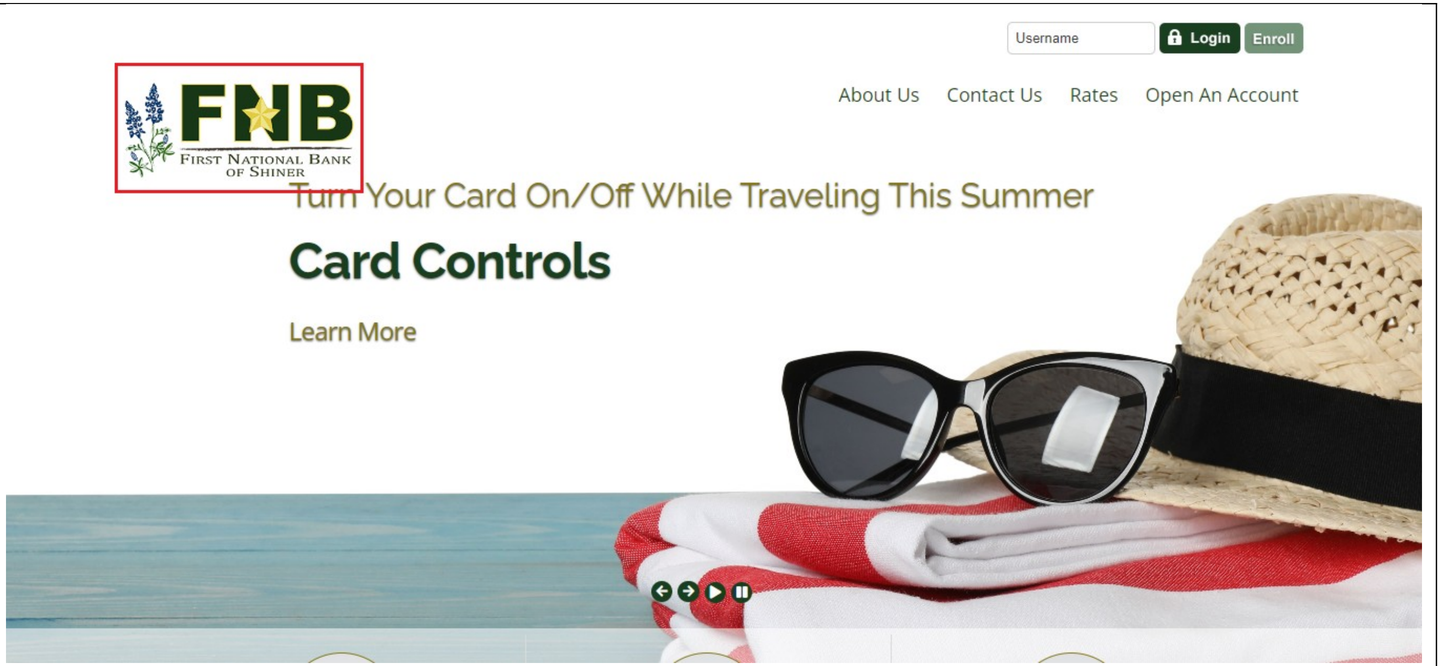
	There is also a decryption function D that takes a ciphertext and a decryption key K_D, and reproduces the plaintext message. $D(C, K_D) = P$ In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster. In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B's encryption key, encrypt a message with it, and send the result to person B. Only someone with B's decryption key, namely only B, can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q, compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de \equiv 1 \pmod{\varphi(n)}$. we know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$. Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is $\boxed{c = m^e \bmod n.}$ $\boxed{\text{First encryption}}$ To decrypt, we compute $c^d \bmod n$ to obtain $c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$ The result of Exercise 3.13 tells us that $\boxed{m \equiv m^{1+k\varphi(n)} \pmod{n},}$ $\boxed{\text{First decryption}}$ https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

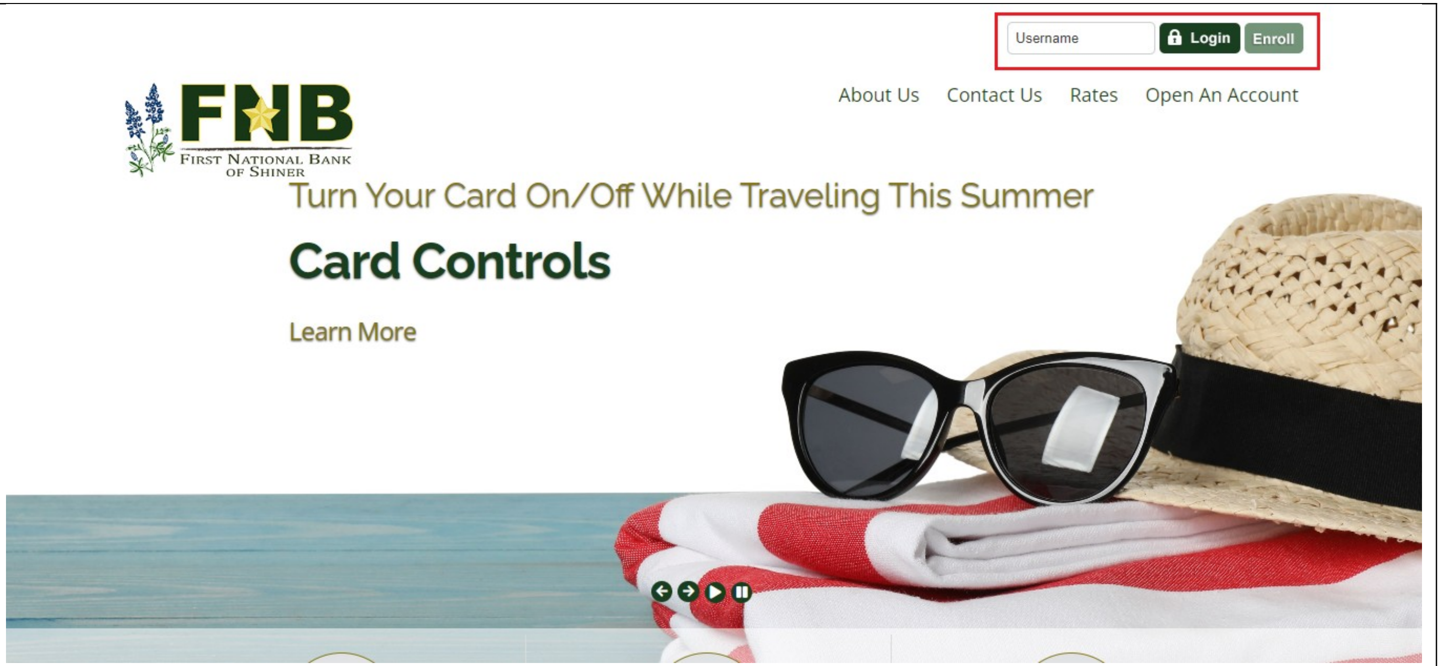
	The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m,h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

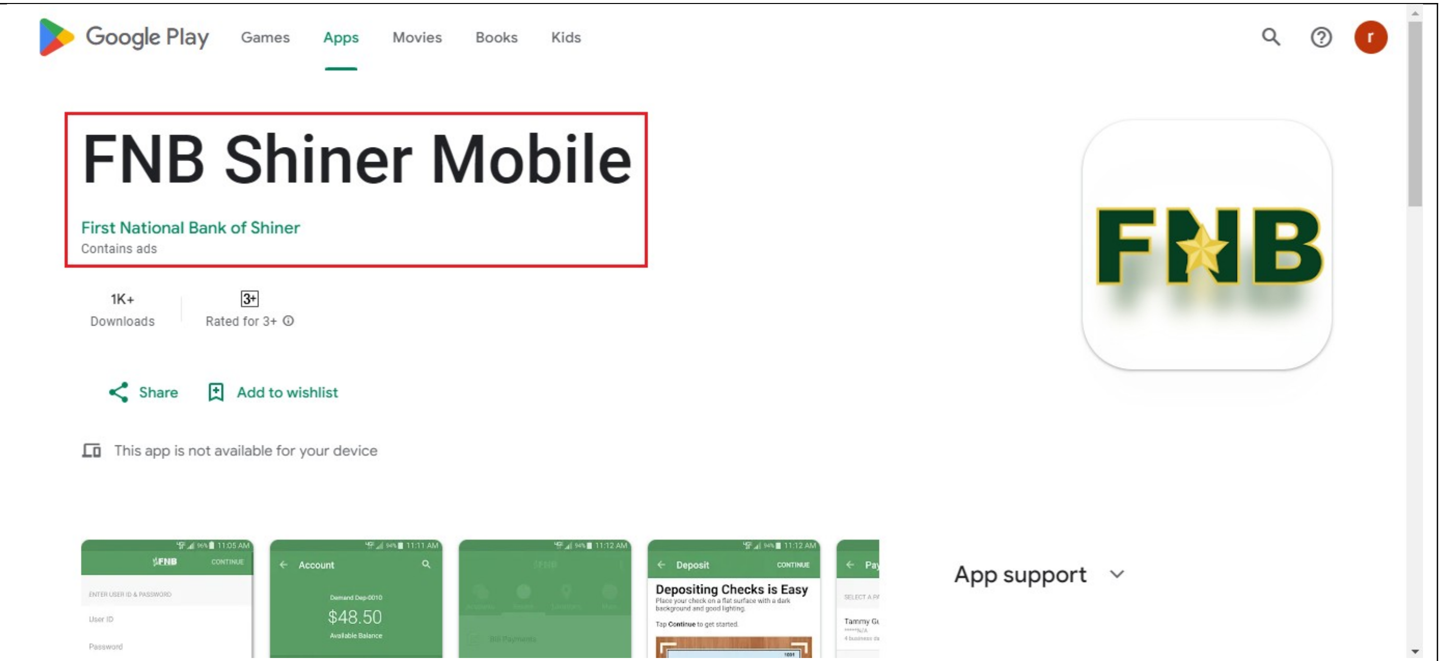
	This specification defines the following cipher suites for use with TLS 1.3. ``` +------------------------------+-------------+ | Description | Value | +------------------------------+-------------+ | TLS_AES_128_GCM_SHA256 | {0x13,0x01} | | | | | TLS_AES_256_GCM_SHA384 | {0x13,0x02} | | | | | TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} | | | | | TLS_AES_128_CCM_SHA256 | {0x13,0x04} | | | | | TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} | +------------------------------+-------------+ ``` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	2.2. Authenticated Decryption The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure). https://datatracker.ietf.org/doc/html/rfc5116 The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1
5. The method of claim 4, wherein the key is contained in a key data structure.	The standard utilized by the accused instrumentality practices the method such that the key (e.g., private key, Key K, etc.) is contained in a key data structure (e.g., data structure).

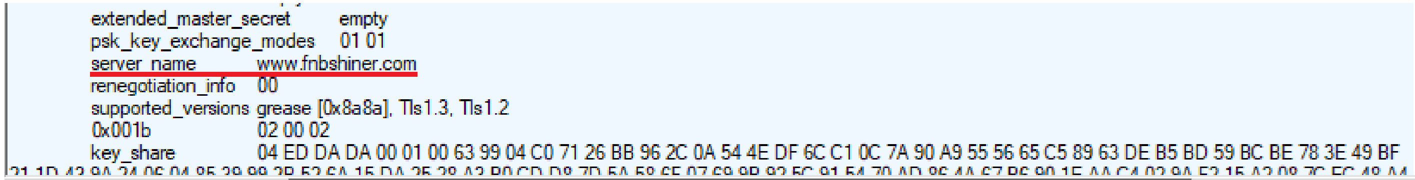


https://www.fnbshiner.com/

	 https://www.fnbshiner.com/



https://play.google.com/store/apps/details?id=com.mfoundry.mb.android.mb_113106833&hl=en_US



Source: Fiddler Capture

The accused instrumentality utilizes a server to establish a secure TLS communication with a client. The server must comprise a memory storage and store data according to a data structure to implement the standard efficiently.

	Tech Accelerator **Server hardware guide: Architecture, products and management** **3. Random access memory** RAM is the main type of memory in a computing system. RAM holds the software instructions and data needed by the processor, along with any output from the processor, such as data to be moved to a storage device. Thus, RAM works very closely with the processor and must match the processor's incredible speed and performance. This kind of fast memory is usually termed dynamic RAM, and several DRAM variations are available for servers. https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms

	Tech Accelerator **Server hardware guide: Architecture, products and management** **4. Hard disk drive** This hardware is responsible for reading, writing and positioning of the hard disk, which is one technology for data storage on server hardware. Developed at IBM in 1953, the hard disk drive (HDD) has evolved over time from the size of a refrigerator to the standard 2.5-inch and 3.5-inch form factors. https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms

	A data structure is a specialized format for organizing, processing, retrieving and storing data. There are several basic and advanced types of data structures, all designed to arrange data to suit a specific purpose. Data structures make it easy for users to access and work with the data they need in appropriate ways. Most importantly, data structures frame the organization of information so that machines and humans can better understand it. In computer science and computer programming, a data structure may be selected or designed to store data for the purpose of using it with various algorithms. In some cases, the algorithm's basic operations are tightly coupled to the data structure's design. Each data structure contains information about the data values, relationships between the data and -- in some cases -- functions that can be applied to the data. https://www.techtarget.com/searchdatamanagement/definition/data-structure As shown below, the server comprises a memory storage to store messages for establishing secure TLS communication. the standard discloses multiple signature encryption algorithms for a first encryption and multiple AEAD encryption algorithms for the second encryption. A signature decryption algorithm utilizes a private key for decrypting the first bitstream encrypted with the signature encryption and an AEAD decryption algorithm uses a key K for decrypting the second bitstream encrypted with the AEAD encryption. Both the decryption techniques are decrypting using their respective associated keys. A server must have a storage to store information pertaining to these algorithms and their corresponding keys such as private key, Key K, etc., to establish secure TLS communication with a client.

	Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT. https://datatracker.ietf.org/doc/html/rfc8446#

	The "extension_data" field of these extensions contains a SignatureSchemeList value: ``` enum { /* RSASSA-PKCS1-v1_5 algorithms */ rsa_pkcs1_sha256(0x0401), rsa_pkcs1_sha384(0x0501), rsa_pkcs1_sha512(0x0601), /* ECDSA algorithms */ ecdsa_secp256r1_sha256(0x0403), ecdsa_secp384r1_sha384(0x0503), ecdsa_secp521r1_sha512(0x0603), /* RSASSA-PSS algorithms with public key OID rsaEncryption */ rsa_pss_rsae_sha256(0x0804), rsa_pss_rsae_sha384(0x0805), rsa_pss_rsae_sha512(0x0806), /* EdDSA algorithms */ ed25519(0x0807), ed448(0x0808), /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */ rsa_pss_pss_sha256(0x0809), rsa_pss_pss_sha384(0x080a), rsa_pss_pss_sha512(0x080b), ``` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	There is also a decryption function D that takes a ciphertext and a decryption key K_D, and reproduces the plaintext message. $D(C, K_D) = P$ In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster. In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B's encryption key, encrypt a message with it, and send the result to person B. Only someone with B's decryption key, namely only B, can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q, compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de = 1 \pmod{\varphi(n)}$. we know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$. Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is $\boxed{c = m^e \bmod n.}$ First encryption To decrypt, we compute $c^d \bmod n$ to obtain $c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$ The result of Exercise 3.13 tells us that $\boxed{m \equiv m^{1+k\varphi(n)} \pmod{n},}$ First decryption https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions `MD5`, `SHA-1`, and `SHA-256`. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	This specification defines the following cipher suites for use with TLS 1.3. | Description | Value | | TLS_AES_128_GCM_SHA256 | {0x13,0x01} | | TLS_AES_256_GCM_SHA384 | {0x13,0x02} | | TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} | | TLS_AES_128_CCM_SHA256 | {0x13,0x04} | | TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} | https://datatracker.ietf.org/doc/html/rfc8446#section-1

	2.2. Authenticated Decryption The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure). https://datatracker.ietf.org/doc/html/rfc5116 The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1
11. The method of claim 3, wherein each encryption algorithm is a symmetric key system or an asymmetric key system.	The standard practices the method such that each encryption algorithm (e.g., signature encryption algorithm i.e., SHA256RSA, etc., and AEAD encryption algorithm i.e., TLS_AES_256_GCM_SHA384, etc.) is a symmetric key system (e.g., AEAD encryption algorithm, etc.) or an asymmetric key system (e.g., signature encryption algorithm). As shown below, the server comprises a memory storage to store messages for

	establishing secure TLS communication. the standard discloses multiple signature encryption algorithms for a first encryption and multiple AEAD encryption algorithms for the second encryption. A signature decryption algorithm utilizes a private key for decrypting the first bitstream encrypted with the signature encryption and an AEAD decryption algorithm uses a key K for decrypting the second bitstream encrypted with the AEAD encryption. The standard defines the signature encryption algorithm as an asymmetric cryptography algorithm and the AEAD encryption algorithm as the symmetric cryptography algorithm. Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT. https://datatracker.ietf.org/doc/html/rfc8446# Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated. Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], the Elliptic Curve Digital Signature Algorithm (ECDSA) [ECDSA], or the Edwards-Curve Digital Signature Algorithm (EdDSA) [RFC8032]) or a symmetric pre-shared key (PSK). https://datatracker.ietf.org/doc/html/rfc8446#section-4

	cipher_suites: A list of the symmetric cipher options supported by the client, specifically the record protection algorithm (including secret key length) and a hash to be used with HKDF, in descending order of client preference. Values are defined in Appendix B.4. If the list contains cipher suites that the server does not recognize, support, or wish to use, the server MUST ignore those cipher suites and process the remaining ones as usual. If the client is attempting a PSK key establishment, it SHOULD advertise at least one cipher suite indicating a Hash associated with the PSK. https://datatracker.ietf.org/doc/html/rfc8446#section-4

	The "extension_data" field of these extensions contains a SignatureSchemeList value: ``` enum { /* RSASSA-PKCS1-v1_5 algorithms */ rsa_pkcs1_sha256(0x0401), rsa_pkcs1_sha384(0x0501), rsa_pkcs1_sha512(0x0601), /* ECDSA algorithms */ ecdsa_secp256r1_sha256(0x0403), ecdsa_secp384r1_sha384(0x0503), ecdsa_secp521r1_sha512(0x0603), /* RSASSA-PSS algorithms with public key OID rsaEncryption */ rsa_pss_rsae_sha256(0x0804), rsa_pss_rsae_sha384(0x0805), rsa_pss_rsae_sha512(0x0806), /* EdDSA algorithms */ ed25519(0x0807), ed448(0x0808), /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */ rsa_pss_pss_sha256(0x0809), rsa_pss_pss_sha384(0x080a), rsa_pss_pss_sha512(0x080b), ``` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	There is also a decryption function D that takes a ciphertext and a decryption key K_D, and reproduces the plaintext message. $D(C, K_D) = P$ In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster. In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B's encryption key, encrypt a message with it, and send the result to person B. Only someone with B's decryption key, namely only B, can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

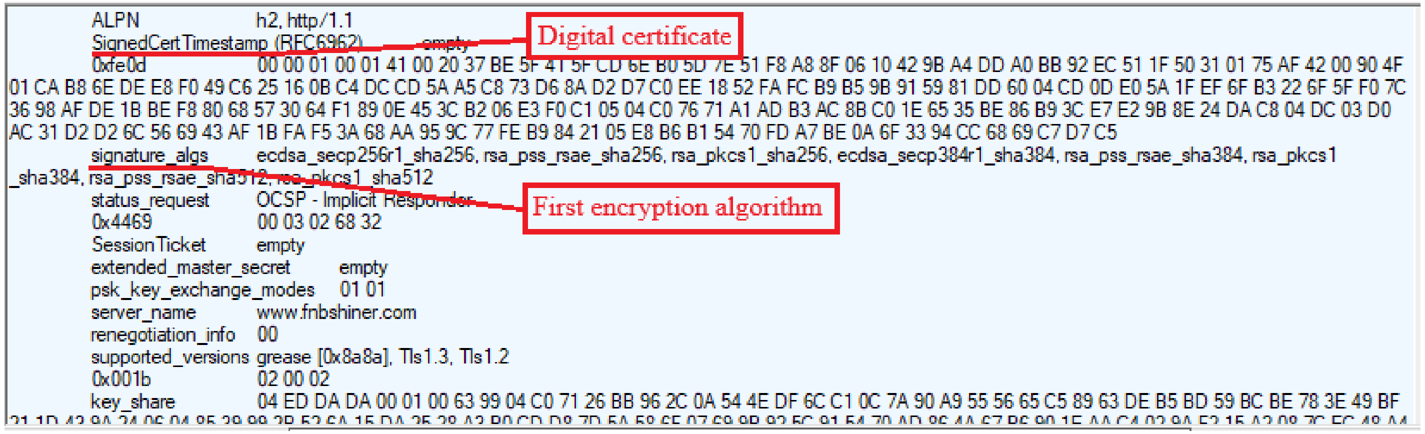
	The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m,h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	This specification defines the following cipher suites for use with TLS 1.3. | Description | Value | | TLS_AES_128_GCM_SHA256 | {0x13,0x01} | | TLS_AES_256_GCM_SHA384 | {0x13,0x02} | | TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} | | TLS_AES_128_CCM_SHA256 | {0x13,0x04} | | TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} | https://datatracker.ietf.org/doc/html/rfc8446#section-1

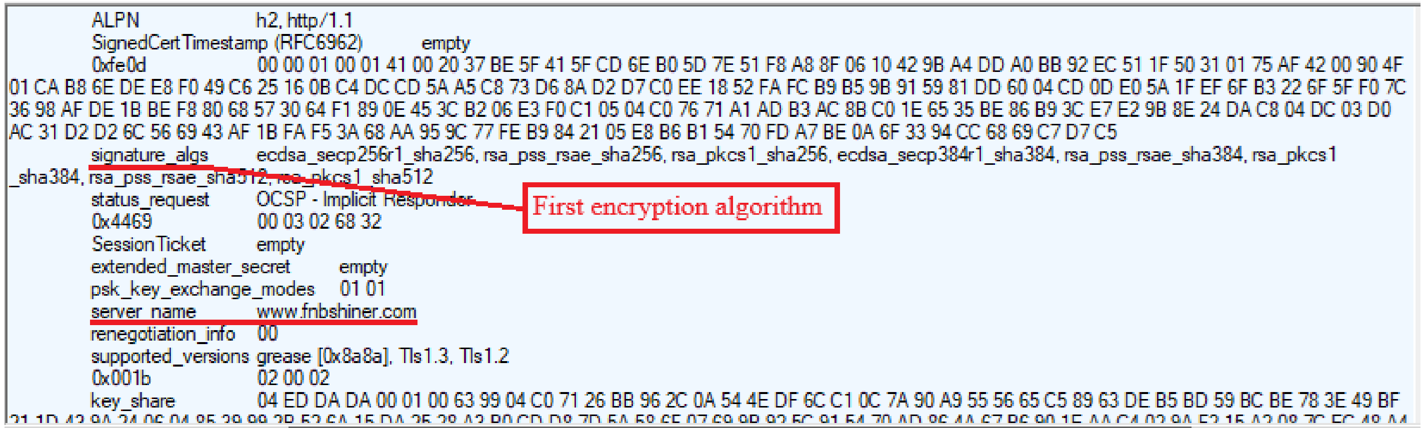
	The authenticated encryption operation has four inputs, each of which is an octet string: A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom. A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in Section 3.2, and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces. A plaintext P, which contains the data to be encrypted and authenticated. The associated data A, which contains the data to be authenticated, but not encrypted. https://datatracker.ietf.org/doc/html/rfc5116

	2.2. Authenticated Decryption The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure). https://datatracker.ietf.org/doc/html/rfc5116 The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1
12. The method of claim 3, further comprising associating a first Message Authentication Code (MAC) or first digital signature with each	The standard practices associating a first Message Authentication Code (MAC) (e.g., message authentication code with hashing function) or first digital signature with each encrypted bit stream (e.g., encrypted bit stream with the signature encryption algorithm i.e., SHA256RSA, etc., and encrypted bitstream with the AEAD encryption algorithm i.e., TLS_AES_256_GCM_SHA384, etc.). As shown below, the standard discloses a hashing function with each of the encryption

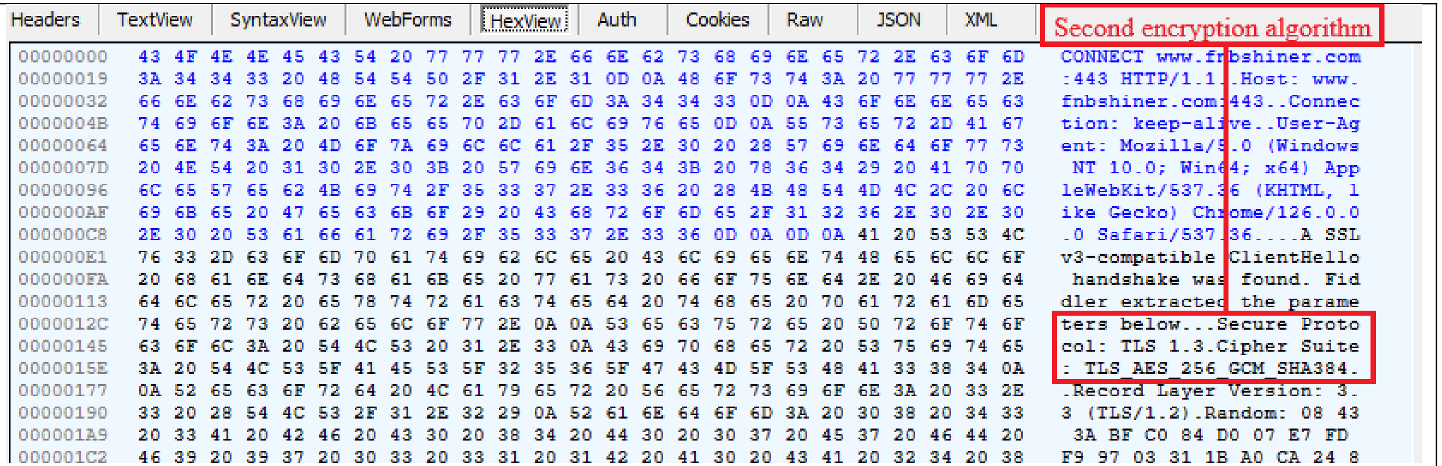
<table>
<tr>
<td>encrypted bit stream.</td>
<td>
algorithm. It performs a message authentication code with the utilized hashing function.



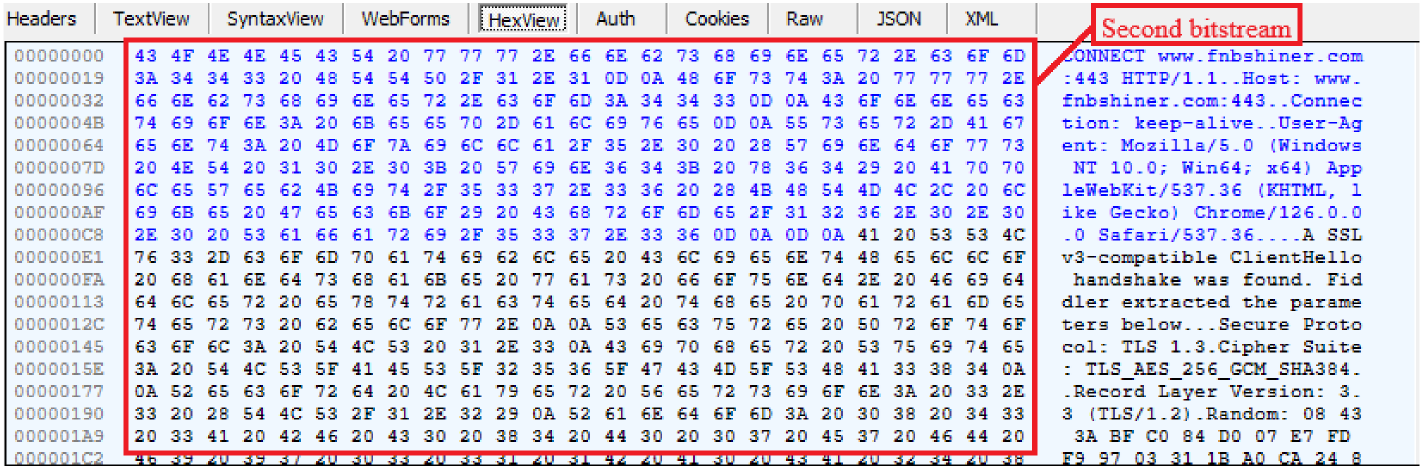
Source: Fiddler Capture



Source: Fiddler Capture
</td>
</tr>
</table>

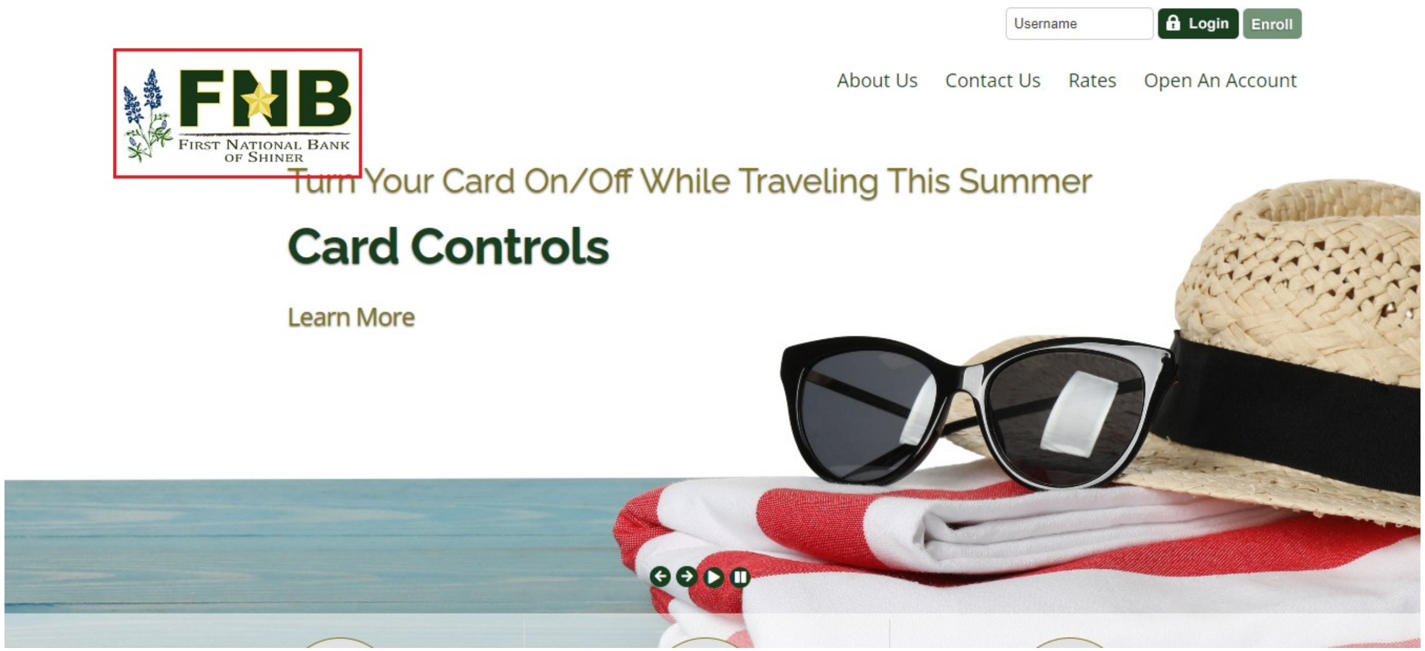


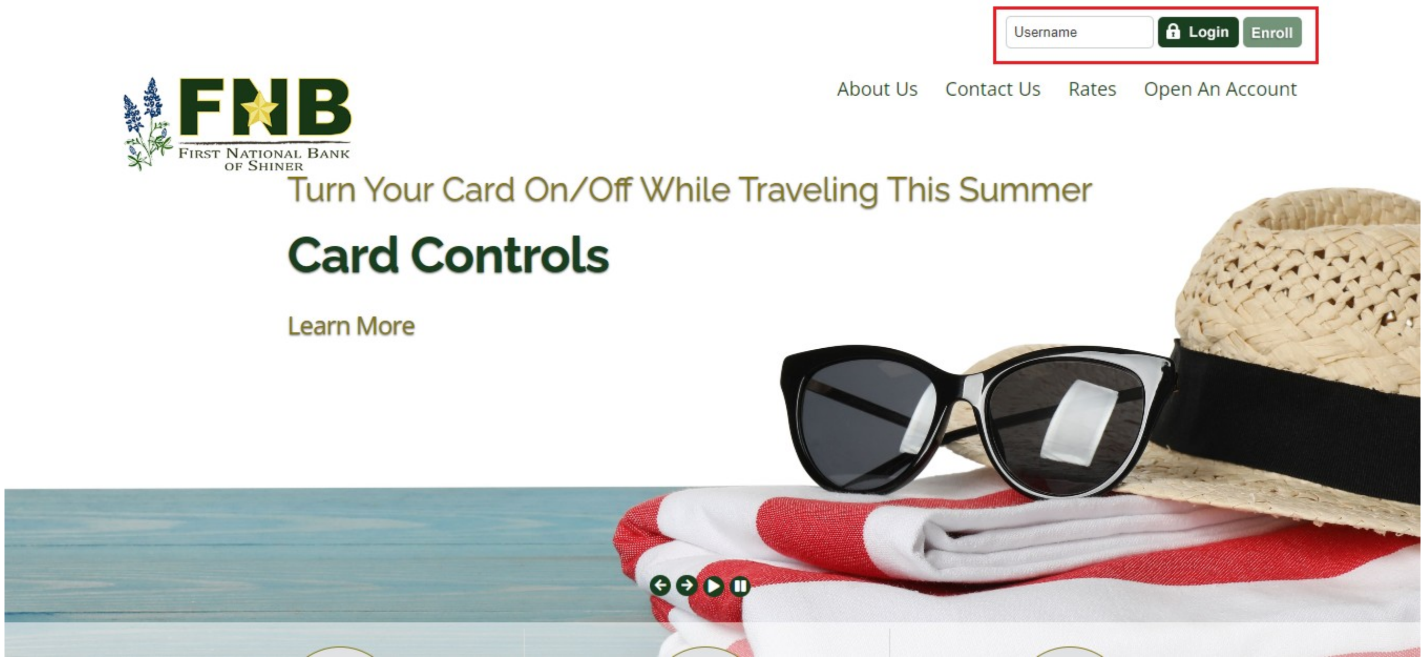
Source: Fiddler Capture

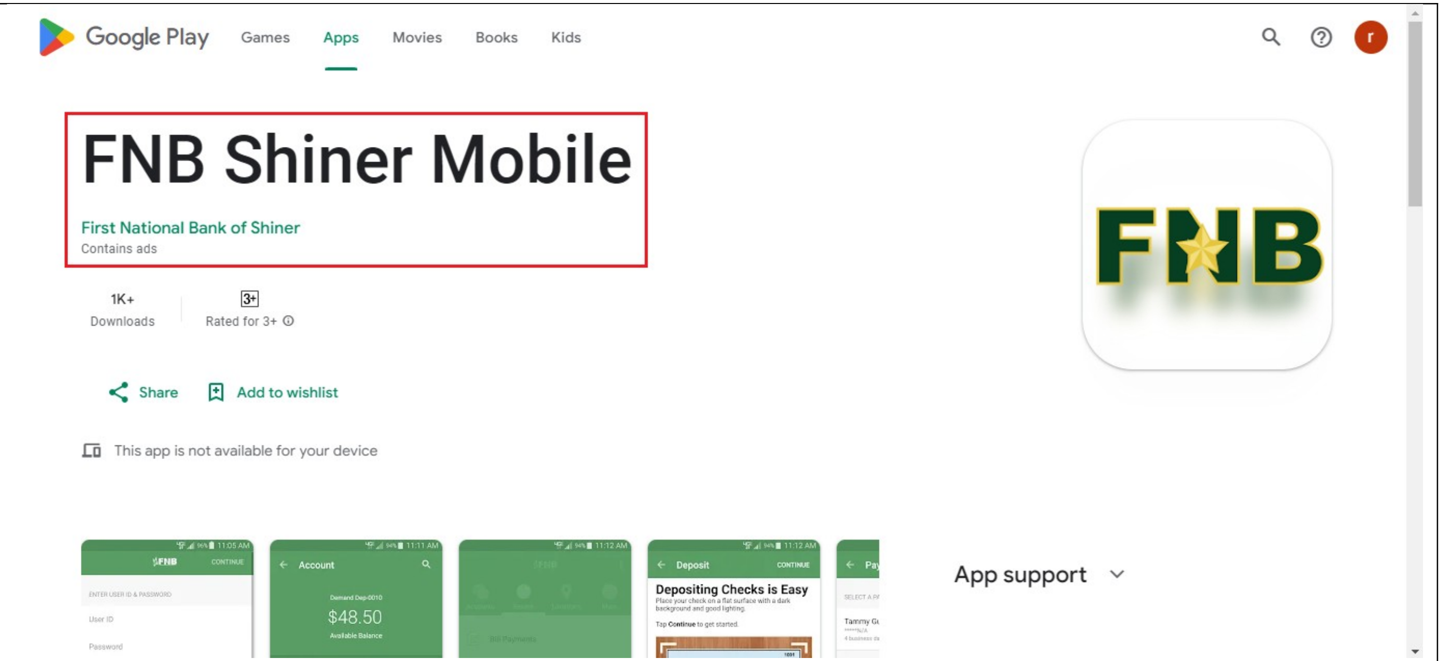


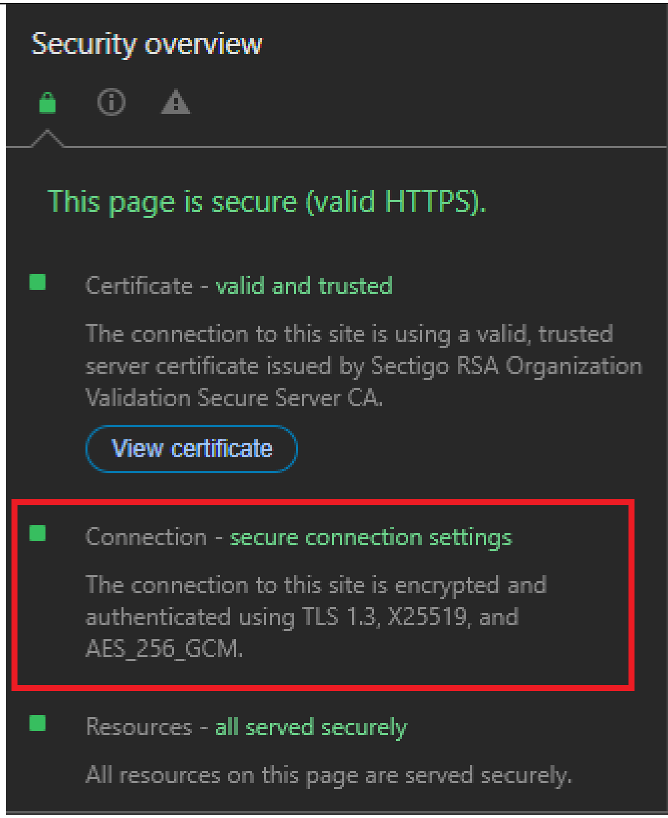
Source: Fiddler Capture

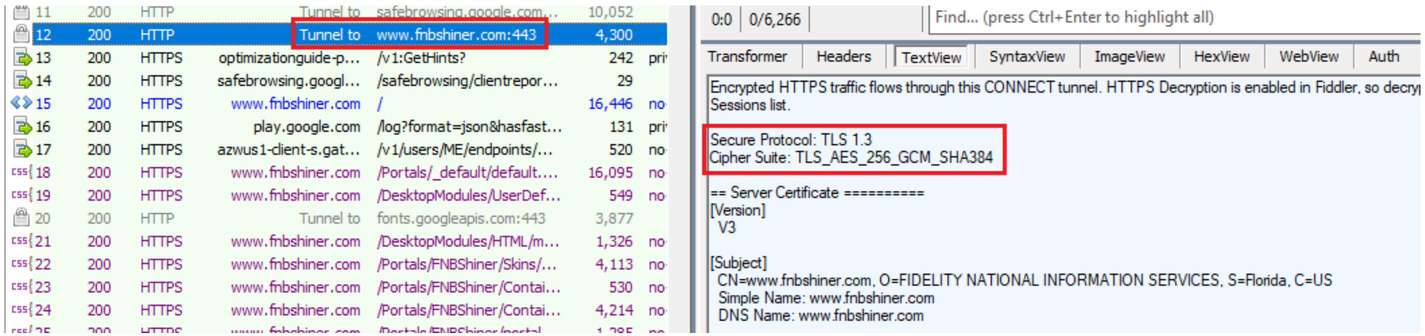
	The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf The list of supported symmetric encryption algorithms has been pruned of all algorithms that are considered legacy. Those that remain are all Authenticated Encryption with Associated Data (AEAD) algorithms. The cipher suite concept has been changed to separate the authentication and key exchange mechanisms from the record protection algorithm (including secret key length) and a hash to be used with both the key derivation function and handshake message authentication code (MAC). https://datatracker.ietf.org/doc/html/rfc8446#section-4
19. A system for a recursive security	The accused instrumentality utilizes a system for a recursive security protocol (e.g., TLS 1.3 security protocol) for protecting digital content (e.g., digital certificate related

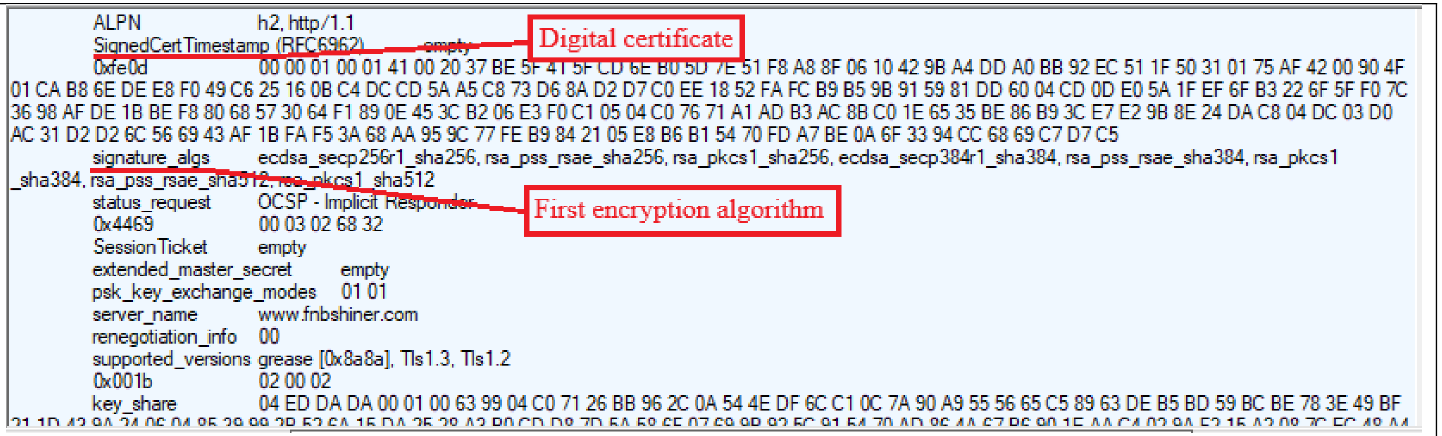
protocol for protecting digital content, comprising a processor to execute instructions and a memory operable to store instructions for performing the steps of:	to the accused instrumentality), comprising a processor (e.g., a processor of the server of the accused instrumentality) to execute instructions and a memory (e.g., a memory of the server of the accused instrumentality) operable to store instructions. The accused instrumentality utilizes TLS 1.3 security protocol (hereinafter "the standard") for communicating content such as digital certificate, application data, etc., with a client. The standard provides a two-level encryption security. It encrypts a plaintext with a first encryption technique and generates a ciphertext. Further, it encrypts the ciphertext with a second encryption technique i.e., recursive encryption security.  https://www.fnbshiner.com/

	 https://www.fnbshiner.com/

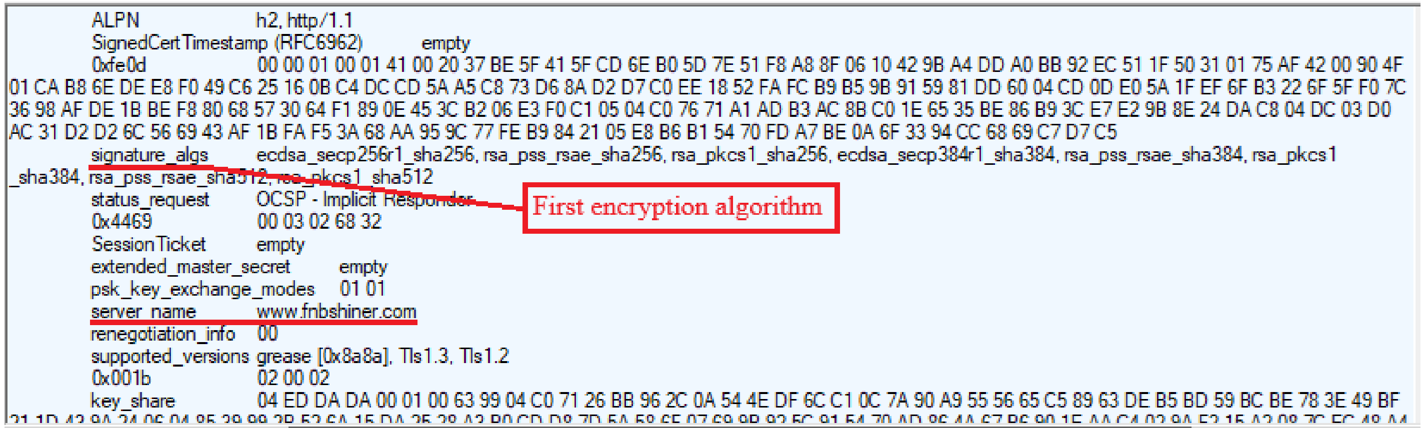
	 https://play.google.com/store/apps/details?id=com.mfoundry.mb.android.mb_113106833&hl=en_US

	 https://www.fnbshiner.com/

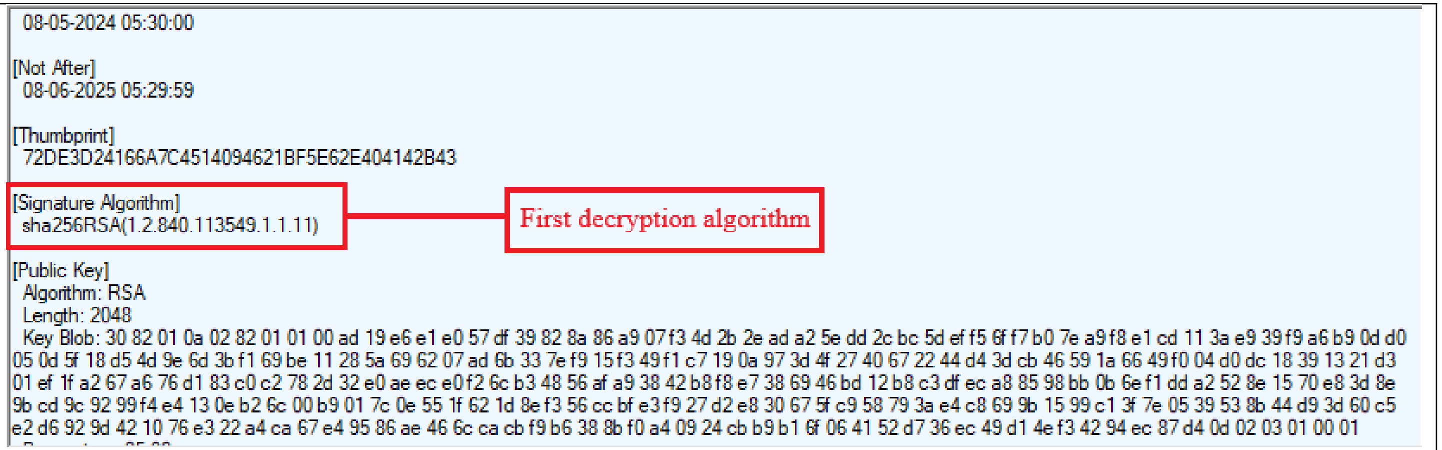
	The Transport Layer Security (TLS) Protocol Version 1.3 Abstract This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery. https://datatracker.ietf.org/doc/html/rfc8446 As shown below, the accused instrumentality utilizes a two-level algorithm security. It utilizes the SHA256RSA encryption algorithm as a first encryption algorithm i.e., signature encryption algorithm and the TLS_AES_256_GCM_SHA384 encryption algorithm as a second encryption algorithm i.e., AEAD encryption algorithm.  *Source: Fiddler Capture*



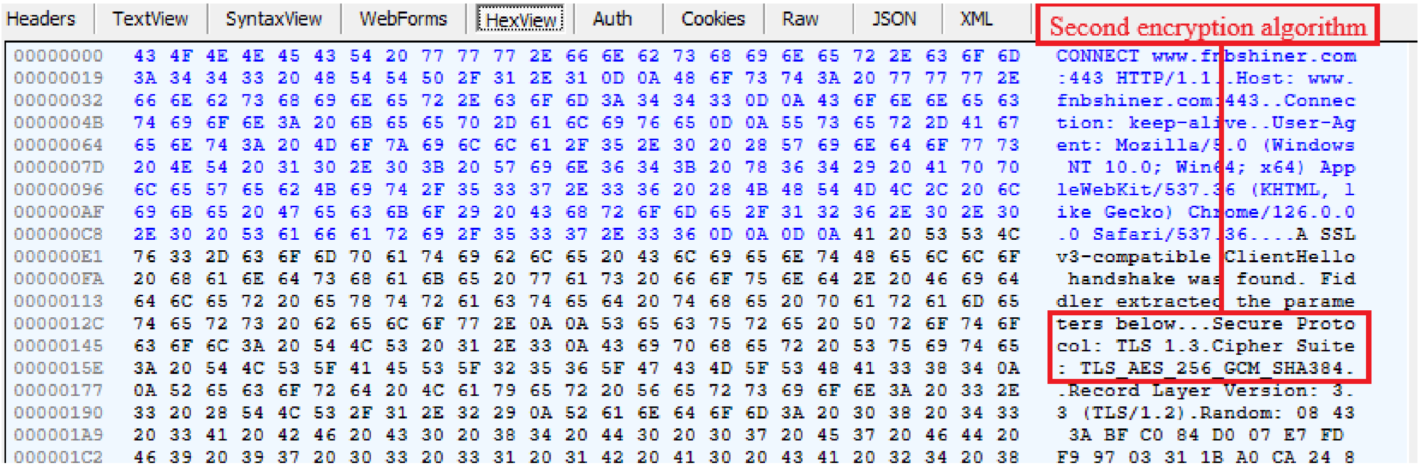
Source: Fiddler Capture



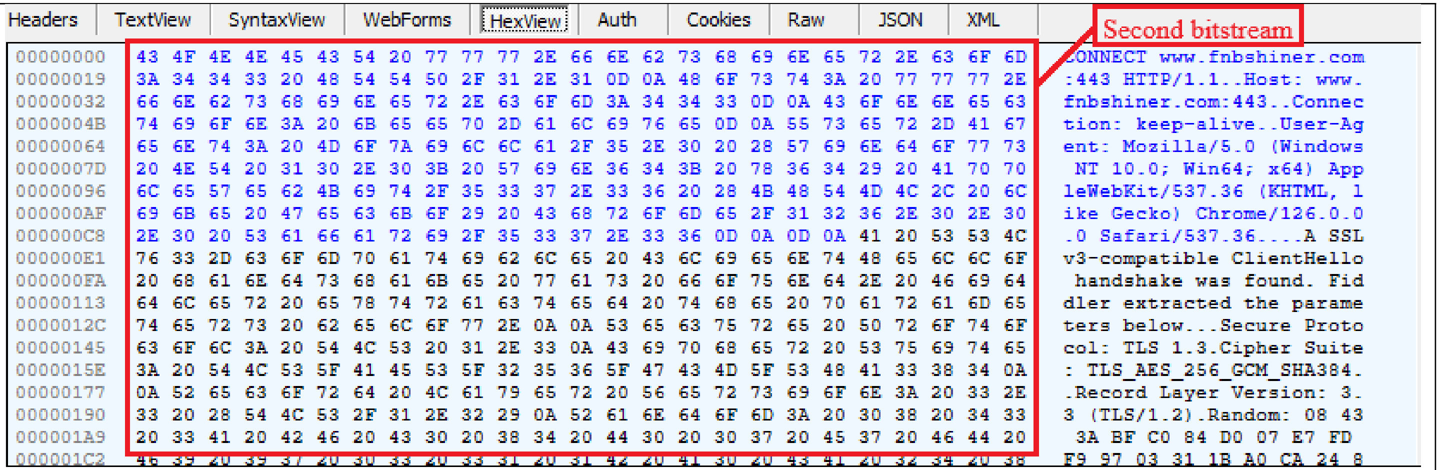
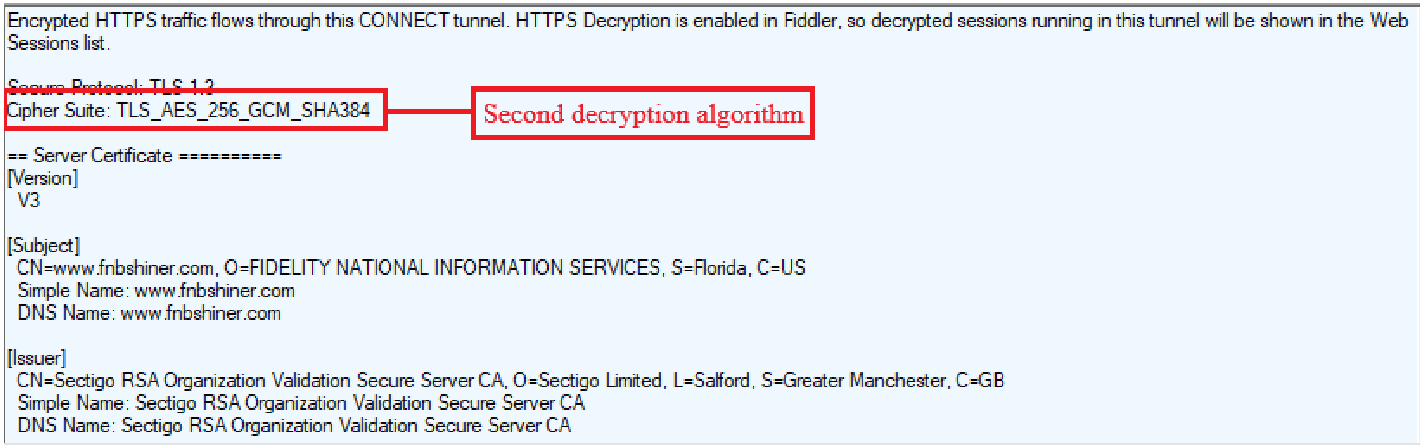
Source: Fiddler Capture

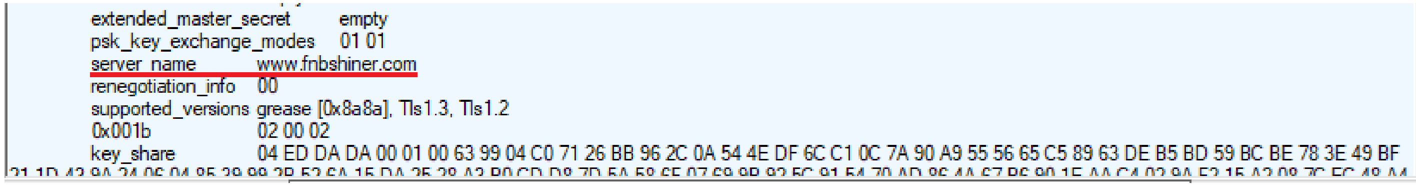


Source: Fiddler Capture



Source: Fiddler Capture

	 *Source: Fiddler Capture*  *Source: Fiddler Capture* As shown below, the server of the accused instrumentality comprises a processor to execute instructions and a memory storage to store instructions for performing the operations defined by the standard.

	 *Source: Fiddler Capture*

	Tech Accelerator **Server hardware guide: Architecture, products and management** **2. Processor** The CPU -- or simply processor -- is a complex micro-circuitry device that serves as the foundation of all computer operations. It supports hundreds of possible commands hardwired into hundreds of millions of transistors to process low-level software instructions -- microcode -- and data and derive a desired logical or mathematical result. The processor works closely with memory, which both holds the software instructions and data to be processed as well as the results or output of those processor operations. https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms

	Tech Accelerator **Server hardware guide: Architecture, products and management** **3. Random access memory** RAM is the main type of memory in a computing system. RAM holds the software instructions and data needed by the processor, along with any output from the processor, such as data to be moved to a storage device. Thus, RAM works very closely with the processor and must match the processor's incredible speed and performance. This kind of fast memory is usually termed dynamic RAM, and several DRAM variations are available for servers. https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms

	Tech Accelerator **Server hardware guide: Architecture, products and management** **4. Hard disk drive** This hardware is responsible for reading, writing and positioning of the hard disk, which is one technology for data storage on server hardware. Developed at IBM in 1953, the hard disk drive (HDD) has evolved over time from the size of a refrigerator to the standard 2.5-inch and 3.5-inch form factors. https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT.

	https://datatracker.ietf.org/doc/html/rfc8446# The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm. As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

	5. Record Protocol The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients. TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see Appendix D.4). https://datatracker.ietf.org/doc/html/rfc8446#section-1 **2. Protocol Overview** Negotiating encryption algorithm The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic. https://datatracker.ietf.org/doc/html/rfc8446

	TLS consists of two primary components: - A handshake protocol (Section 4) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack. Negotiating encryption algos - A record protocol (Section 5) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys. https://datatracker.ietf.org/doc/html/rfc8446 All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4. Authentication Messages As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [sender]_handshake_traffic_secret. https://datatracker.ietf.org/doc/html/rfc8446#section-1

Figure 1 below shows the basic full TLS handshake:

```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

Digital Content

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.1.1. Cryptographic Negotiation In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello: - A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports. Second encryption - A "supported_groups" (Section 4.2.7) extension which indicates the (EC)DHE groups which the client supports and a "key_share" (Section 4.2.8) extension which contains (EC)DHE shares for some or all of these groups. - A "signature_algorithms" (Section 4.2.3) extension which indicates the signature algorithms which the client can accept. A "signature_algorithms_cert" extension (Section 4.2.3) may also be added to indicate certificate-specific signature algorithms. First encryption - A "pre_shared_key" (Section 4.2.11) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" (Section 4.2.9) extension which indicates the key exchange modes that may be used with PSKs. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2. Certificate This message conveys the endpoint's certificate chain to the peer. The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK). The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2.3. Client Certificate Selection The following rules apply to certificates sent by the client: - The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]). - If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs. - The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS. - If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.3. Certificate Verify This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message. Structure of this message: `struct {` `   SignatureScheme algorithm;` `   opaque signature<0..2^16-1>;` `} CertificateVerify;` The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely: https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	4.3.2. Certificate Request A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions. Structure of this message: `struct {` `opaque certificate_request_context<0..2^8-1>;` `Extension extensions<2..2^16-1>;` `} CertificateRequest;` https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages. extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions. https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message. In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.2.3. Signature Algorithms TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2). The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	The "extension_data" field of these extensions contains a SignatureSchemeList value: ``` enum { /* RSASSA-PKCS1-v1_5 algorithms */ rsa_pkcs1_sha256(0x0401), rsa_pkcs1_sha384(0x0501), rsa_pkcs1_sha512(0x0601), /* ECDSA algorithms */ ecdsa_secp256r1_sha256(0x0403), ecdsa_secp384r1_sha384(0x0503), ecdsa_secp521r1_sha512(0x0603), /* RSASSA-PSS algorithms with public key OID rsaEncryption */ rsa_pss_rsae_sha256(0x0804), rsa_pss_rsae_sha384(0x0805), rsa_pss_rsae_sha512(0x0806), /* EdDSA algorithms */ ed25519(0x0807), ed448(0x0808), /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */ rsa_pss_pss_sha256(0x0809), rsa_pss_pss_sha384(0x080a), rsa_pss_pss_sha512(0x080b), ``` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	Introduction The primary goal of TLS is to provide a secure channel between two communicating peers; the only requirement from the underlying transport is a reliable, in-order data stream. Specifically, the secure channel should provide the following properties: First encryption - Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated. Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], the Elliptic Curve Digital Signature Algorithm (ECDSA) [ECDSA], or the Edwards-Curve Digital Signature Algorithm (EdDSA) [RFC8032]) or a symmetric pre-shared key (PSK). - Confidentiality: Data sent over the channel after establishment is only visible to the endpoints. TLS does not hide the length of the data it transmits, though endpoints are able to pad TLS records in order to obscure lengths and improve protection against traffic analysis techniques. - Integrity: Data sent over the channel after establishment cannot be modified by attackers without detection. https://datatracker.ietf.org/doc/html/rfc8446

	5.1. Record Layer The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^14 bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2. Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that: - Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **5.2. Record Payload Protection** The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding. https://datatracker.ietf.org/doc/html/rfc8446#section-1

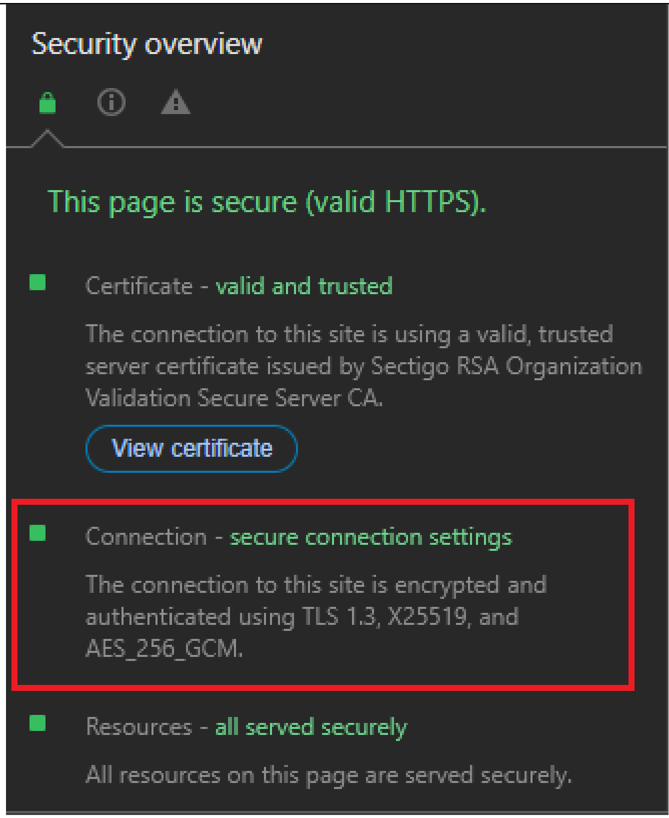
	AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see Section 5.3), and the additional data input is the record header. I.e., `additional_data = TLSCiphertext.opaque_type ||` `TLSCiphertext.legacy_record_version ||` `TLSCiphertext.length` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case. https://datatracker.ietf.org/doc/html/rfc5116

	2.1. Authenticated Encryption The authenticated encryption operation has four inputs, each of which is an octet string: A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom. A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in Section 3.2, and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces. A plaintext P, which contains the data to be encrypted and authenticated. The associated data A, which contains the data to be authenticated, but not encrypted. https://datatracker.ietf.org/doc/html/rfc5116

	The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1 Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [RFC5116], Section 4). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows: https://datatracker.ietf.org/doc/html/rfc8446#section-1

	This specification defines the following cipher suites for use with TLS 1.3. | Description | Value | | TLS_AES_128_GCM_SHA256 | {0x13,0x01} | | TLS_AES_256_GCM_SHA384 | {0x13,0x02} | | TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} | | TLS_AES_128_CCM_SHA256 | {0x13,0x04} | | TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} | https://datatracker.ietf.org/doc/html/rfc8446#section-1
encrypting a bit stream with a first encryption algorithm;	The standard practices encrypting a bitstream (e.g., bitstream of digital certificate) with a first encryption algorithm (e.g., signature encryption algorithm i.e., SHA256RSA encryption algorithm). The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA encryption algorithm) and generates a ciphertext.

	 https://www.fnbshiner.com/

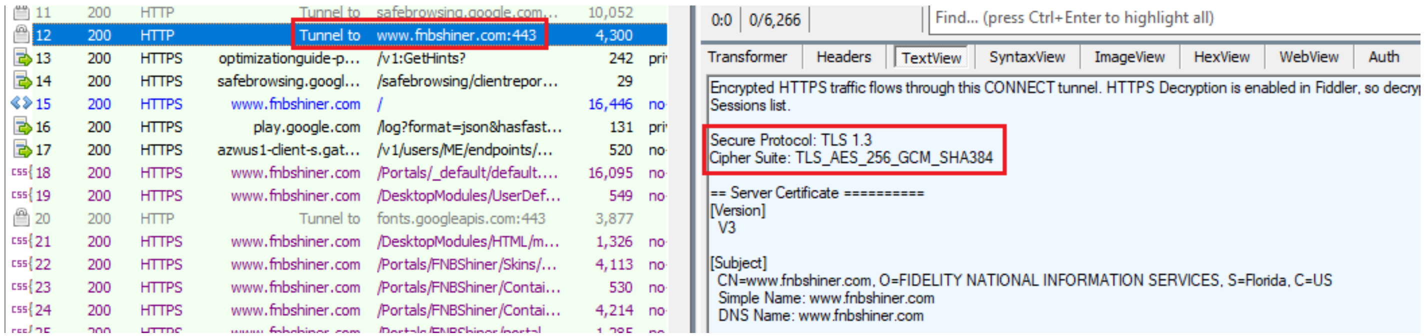
The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

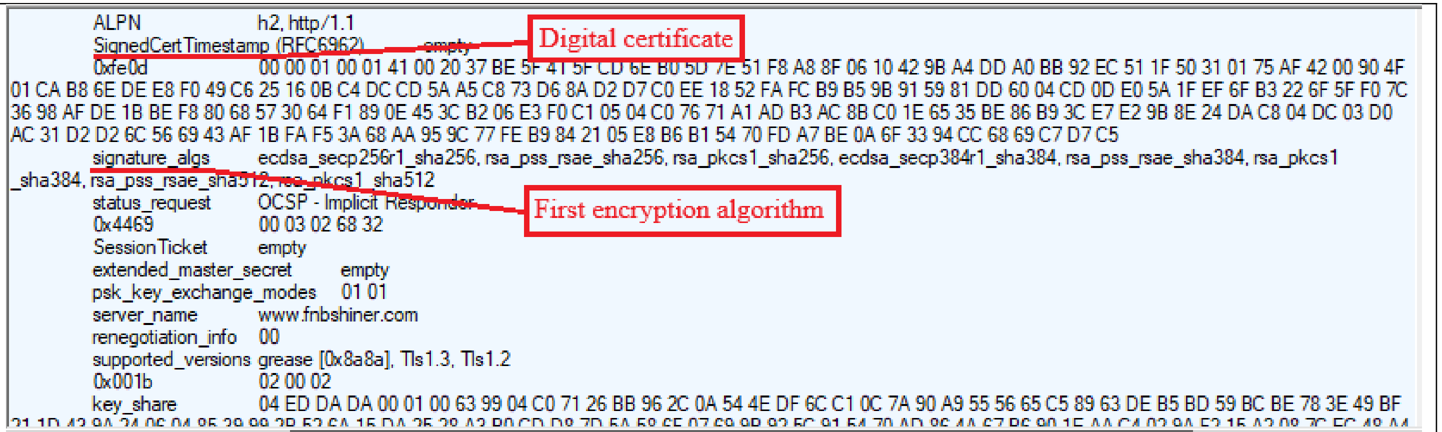
This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

https://datatracker.ietf.org/doc/html/rfc8446

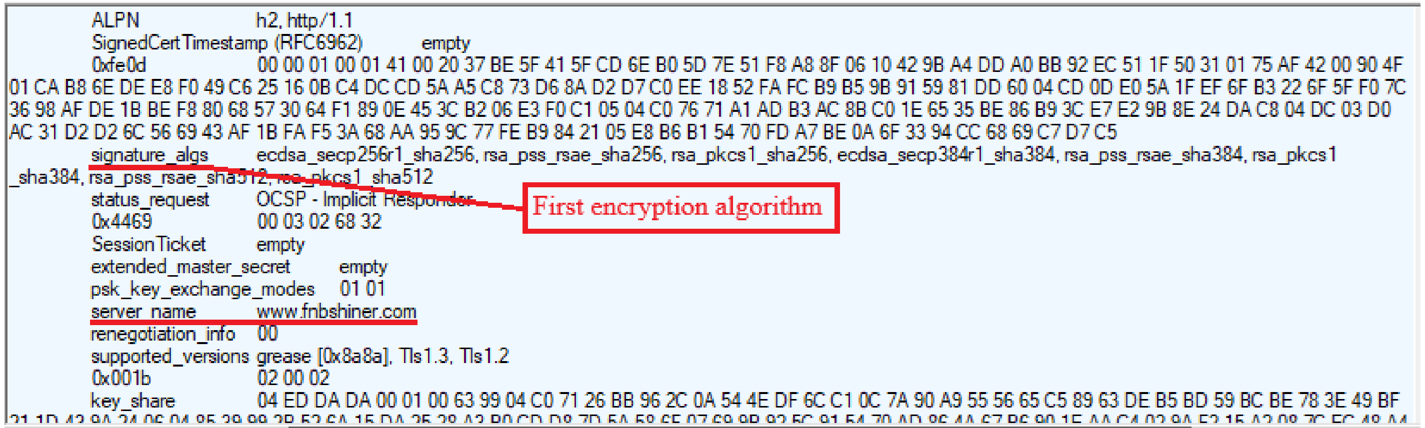
As shown below, the accused instrumentality discloses the signature encryption algorithm.



Source: Fiddler Capture



Source: Fiddler Capture



Source: Fiddler Capture

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are

	encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm. As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server. **5. Record Protocol** The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients. TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see Appendix D.4). https://datatracker.ietf.org/doc/html/rfc8446#section-1

	2. Protocol Overview Negotiating encryption algorithm The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic. https://datatracker.ietf.org/doc/html/rfc8446 TLS consists of two primary components: - A handshake protocol (Section 4) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack. Negotiating encryption algos - A record protocol (Section 5) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys. https://datatracker.ietf.org/doc/html/rfc8446

	All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **4.4. Authentication Messages** As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [sender]_handshake_traffic_secret. https://datatracker.ietf.org/doc/html/rfc8446#section-1

```
   Figure 1 below shows the basic full TLS handshake:

       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

Digital Content

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.1.1. Cryptographic Negotiation In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello: - A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports. - A "supported_groups" (Section 4.2.7) extension which indicates the (EC)DHE groups which the client supports and a "key_share" (Section 4.2.8) extension which contains (EC)DHE shares for some or all of these groups. First encryption - A "signature_algorithms" (Section 4.2.3) extension which indicates the signature algorithms which the client can accept. A "signature_algorithms_cert" extension (Section 4.2.3) may also be added to indicate certificate-specific signature algorithms. - A "pre_shared_key" (Section 4.2.11) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" (Section 4.2.9) extension which indicates the key exchange modes that may be used with PSKs. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2. Certificate This message conveys the endpoint's certificate chain to the peer. The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK). The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2.3. Client Certificate Selection The following rules apply to certificates sent by the client: - The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]). - If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs. - The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS. - If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.3. Certificate Verify This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message. Structure of this message: `struct {` `SignatureScheme algorithm;` `opaque signature<0..2^16-1>;` `} CertificateVerify;` The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely: https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	4.3.2. Certificate Request A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions. Structure of this message: `struct {` `    opaque certificate_request_context<0..2^8-1>;` `    Extension extensions<2..2^16-1>;` `} CertificateRequest;` https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages. extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions. https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message. In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.2.3. Signature Algorithms TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2). The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	The "extension_data" field of these extensions contains a SignatureSchemeList value: `enum {` `    /* RSASSA-PKCS1-v1_5 algorithms */` `    rsa_pkcs1_sha256(0x0401),` `    rsa_pkcs1_sha384(0x0501),` `    rsa_pkcs1_sha512(0x0601),` `    /* ECDSA algorithms */` `    ecdsa_secp256r1_sha256(0x0403),` `    ecdsa_secp384r1_sha384(0x0503),` `    ecdsa_secp521r1_sha512(0x0603),` `    /* RSASSA-PSS algorithms with public key OID rsaEncryption */` `    rsa_pss_rsae_sha256(0x0804),` `    rsa_pss_rsae_sha384(0x0805),` `    rsa_pss_rsae_sha512(0x0806),` `    /* EdDSA algorithms */` `    ed25519(0x0807),` `    ed448(0x0808),` `    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */` `    rsa_pss_pss_sha256(0x0809),` `    rsa_pss_pss_sha384(0x080a),` `    rsa_pss_pss_sha512(0x080b),` https://datatracker.ietf.org/doc/html/rfc8446#section-1

<table>
<tr>
<td></td>
<td>

Introduction

The primary goal of TLS is to provide a secure channel between two communicating peers; the only requirement from the underlying transport is a reliable, in-order data stream. Specifically, the secure channel should provide the following properties:

First encryption

- Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated. Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], the Elliptic Curve Digital Signature Algorithm (ECDSA) [ECDSA], or the Edwards-Curve Digital Signature Algorithm (EdDSA) [RFC8032]) or a symmetric pre-shared key (PSK).

- Confidentiality: Data sent over the channel after establishment is only visible to the endpoints. TLS does not hide the length of the data it transmits, though endpoints are able to pad TLS records in order to obscure lengths and improve protection against traffic analysis techniques.

- Integrity: Data sent over the channel after establishment cannot be modified by attackers without detection.

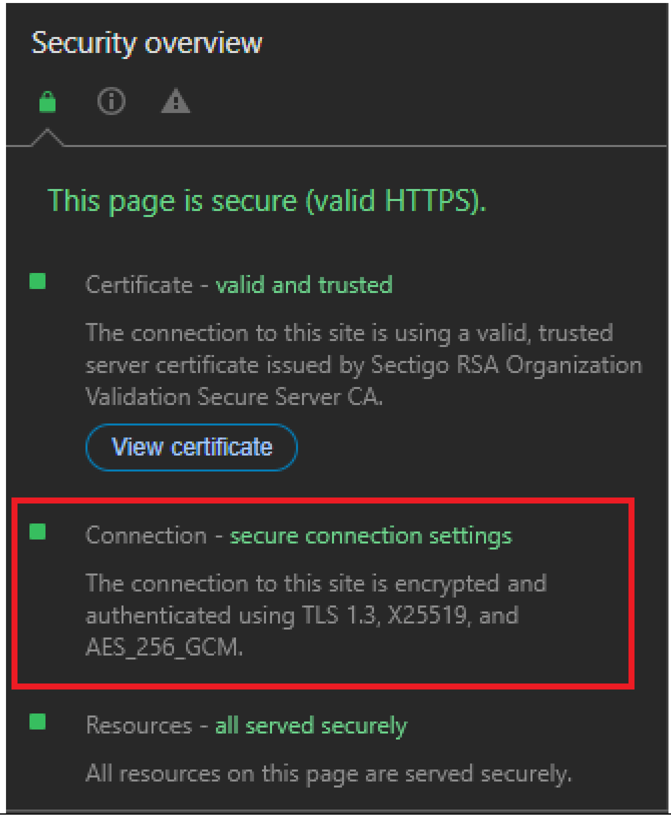
https://datatracker.ietf.org/doc/html/rfc8446

</td>
</tr>
<tr>
<td>associating a first decryption algorithm with the encrypted bit stream;</td>
<td>

The standard practices associating a first decryption algorithm (e.g., signature decryption algorithm i.e., SHA256RSA decryption algorithm) with the encrypted bit stream (e.g., encrypted certificate with signature encryption algorithm).

The standard practices providing a two-level encryption security for data

</td>
</tr>
</table>

	communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA encryption algorithm) and generates a ciphertext. The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it. The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate. 

https://www.fnbshiner.com/

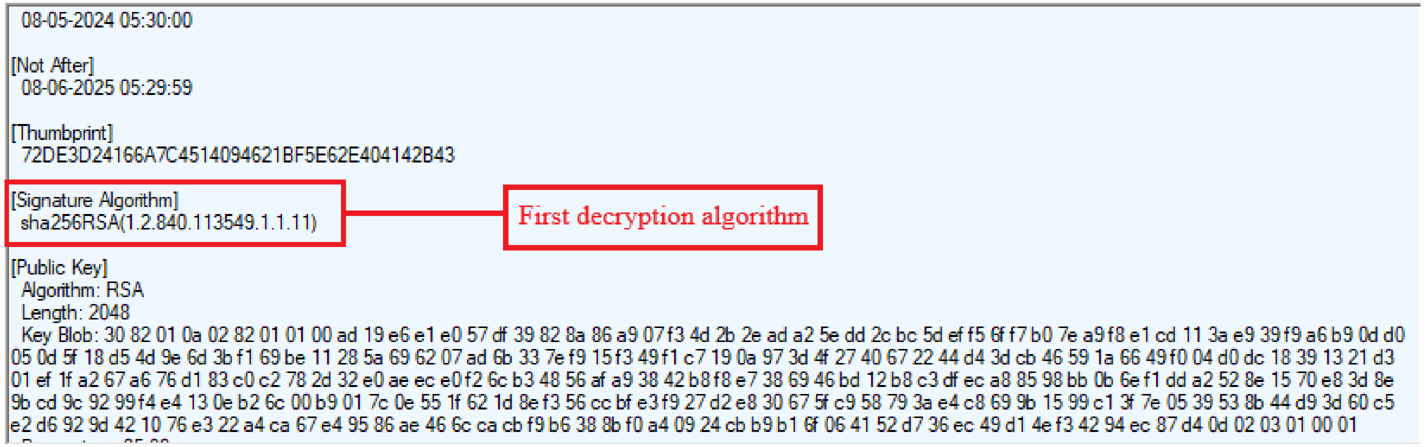
The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

https://datatracker.ietf.org/doc/html/rfc8446

As shown below, the accused instrumentality discloses the signature decryption algorithm.



08-05-2024 05:30:00

[Not After]
08-06-2025 05:29:59

[Thumbprint]
72DE3D24166A7C4514094621BF5E62E404142B43

[Signature Algorithm]
sha256RSA(1.2.840.113549.1.1.11)

First decryption algorithm

[Public Key]
Algorithm: RSA
Length: 2048
Key Blob: 30 82 01 0a 02 82 01 01 00 ad 19 e6 e1 e0 57 df 39 82 8a 86 a9 07 f3 4d 2b 2e ad a2 5e dd 2c bc 5d ef f5 6f f7 b0 7e a9 f8 e1 cd 11 3a e9 39 f9 a6 b9 0d d0 05 0d 5f 18 d5 4d 9e 6d 3b f1 69 be 11 28 5a 69 62 07 ad 6b 33 7e f9 15 f3 49 f1 c7 19 0a 97 3d 4f 27 40 67 22 44 d4 3d cb 46 59 1a 66 49 f0 04 d0 dc 18 39 13 21 d3 01 ef 1f a2 67 a6 76 d1 83 c0 c2 78 2d 32 e0 ae ec e0 f2 6c b3 48 56 af a9 38 42 b8 f8 e7 38 69 46 bd 12 b8 c3 df ec a8 85 98 bb 0b 6e f1 dd a2 52 8e 15 70 e8 3d 8e 9b cd 9c 92 99 f4 e4 13 0e b2 6c 00 b9 01 7c 0e 55 1f 62 1d 8e f3 56 cc bf e3 f9 27 d2 e8 30 67 5f c9 58 79 3a e4 c8 69 9b 15 99 c1 3f 7e 05 39 53 8b 44 d9 3d 60 c5 e2 d6 92 9d 42 10 76 e3 22 a4 ca 67 e4 95 86 ae 46 6c ca cb f9 b6 38 8b f0 a4 09 24 cb b9 b1 6f 06 41 52 d7 36 ec 49 d1 4e f3 42 94 ec 87 d4 0d 02 03 01 00 01

Source: Fiddler Capture

	OID description First decryption algorithm identifier **OID**: {iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs-1(1) sha256WithRSAEncryption(11)} (ASN.1 notation) 1.2.840.113549.1.1.11 (dot notation) /ISO/Member-Body/US/113549/1/1/11 (OID-IRI notation) **Description**: Public-Key Cryptography Standards (PKCS) #1 version 1.5 signature algorithm with Secure Hash Algorithm 256 (SHA256) and Rivest, Shamir and Adleman (RSA) encryption http://oid-info.com/get/1.2.840.113549.1.1.11 `-- When the following OIDs are used in an AlgorithmIdentifier, the` `-- parameters MUST be present and MUST be NULL.` `sha224WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 14 }` `sha256WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 11 }` `sha384WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 12 }` `sha512WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 13 }` https://www.ietf.org/rfc/rfc4055.txt

Figure 1 below shows the basic full TLS handshake:

```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

Digital Content

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.3. Certificate Verify This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message. Structure of this message: `struct {` `    SignatureScheme algorithm;` `    opaque signature<0..2^16-1>;` `} CertificateVerify;` The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely: https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages. extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions. https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	- RSASSA-PSS signature schemes are defined in Section 4.2.3. - The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello. - The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates. https://datatracker.ietf.org/doc/html/rfc8446#section-1 The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm. As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

	5. Record Protocol The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients. TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see Appendix D.4). https://datatracker.ietf.org/doc/html/rfc8446#section-1 **2. Protocol Overview** Negotiating encryption algorithm The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic. https://datatracker.ietf.org/doc/html/rfc8446

	TLS consists of two primary components: - A handshake protocol (Section 4) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack. Negotiating encryption algos - A record protocol (Section 5) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys. https://datatracker.ietf.org/doc/html/rfc8446 All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4. Authentication Messages As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [sender]_handshake_traffic_secret. https://datatracker.ietf.org/doc/html/rfc8446#section-1

Figure 1 below shows the basic full TLS handshake:

Digital Content

```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.1.1. Cryptographic Negotiation In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello: - A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports. - A "supported_groups" (Section 4.2.7) extension which indicates the (EC)DHE groups which the client supports and a "key_share" (Section 4.2.8) extension which contains (EC)DHE shares for some or all of these groups. First encryption - A "signature_algorithms" (Section 4.2.3) extension which indicates the signature algorithms which the client can accept. A "signature_algorithms_cert" extension (Section 4.2.3) may also be added to indicate certificate-specific signature algorithms. - A "pre_shared_key" (Section 4.2.11) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" (Section 4.2.9) extension which indicates the key exchange modes that may be used with PSKs. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2. Certificate This message conveys the endpoint's certificate chain to the peer. The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK). The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2.3. Client Certificate Selection The following rules apply to certificates sent by the client: - The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]). - If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs. - The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS. - If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.3. Certificate Verify This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message. Structure of this message: `struct {` `SignatureScheme algorithm;` `opaque signature<0..2^16-1>;` `} CertificateVerify;` First decryption algorithm information The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely: https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

<table>
<tr>
<td></td>
<td>
4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:
<pre>
   struct {
       opaque certificate_request_context<0..2^8-1>;
       Extension extensions<2..2^16-1>;
   } CertificateRequest;
</pre>
https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2
</td>
</tr>
</table>

	certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages. extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions. https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	- RSASSA-PSS signature schemes are defined in Section 4.2.3. - The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello. - The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates. https://datatracker.ietf.org/doc/html/rfc8446#section-1 If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message. In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.2.3. Signature Algorithms TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2). The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	The "extension_data" field of these extensions contains a SignatureSchemeList value: ``` enum { /* RSASSA-PKCS1-v1_5 algorithms */ rsa_pkcs1_sha256(0x0401), rsa_pkcs1_sha384(0x0501), rsa_pkcs1_sha512(0x0601), /* ECDSA algorithms */ ecdsa_secp256r1_sha256(0x0403), ecdsa_secp384r1_sha384(0x0503), ecdsa_secp521r1_sha512(0x0603), /* RSASSA-PSS algorithms with public key OID rsaEncryption */ rsa_pss_rsae_sha256(0x0804), rsa_pss_rsae_sha384(0x0805), rsa_pss_rsae_sha512(0x0806), /* EdDSA algorithms */ ed25519(0x0807), ed448(0x0808), /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */ rsa_pss_pss_sha256(0x0809), rsa_pss_pss_sha384(0x080a), rsa_pss_pss_sha512(0x080b), ``` https://datatracker.ietf.org/doc/html/rfc8446#section-1 As shown below, the receiving party will be able to decrypt the encrypted message with the provided signature decryption algorithm information i.e., SHA-256 RSA decryption algorithm.

We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q, compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de = 1 \pmod{\varphi(n)}$. we know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$.

Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is

$$\boxed{c = m^e \bmod n.}$$

First encryption

To decrypt, we compute $c^d \bmod n$ to obtain

$$c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$$

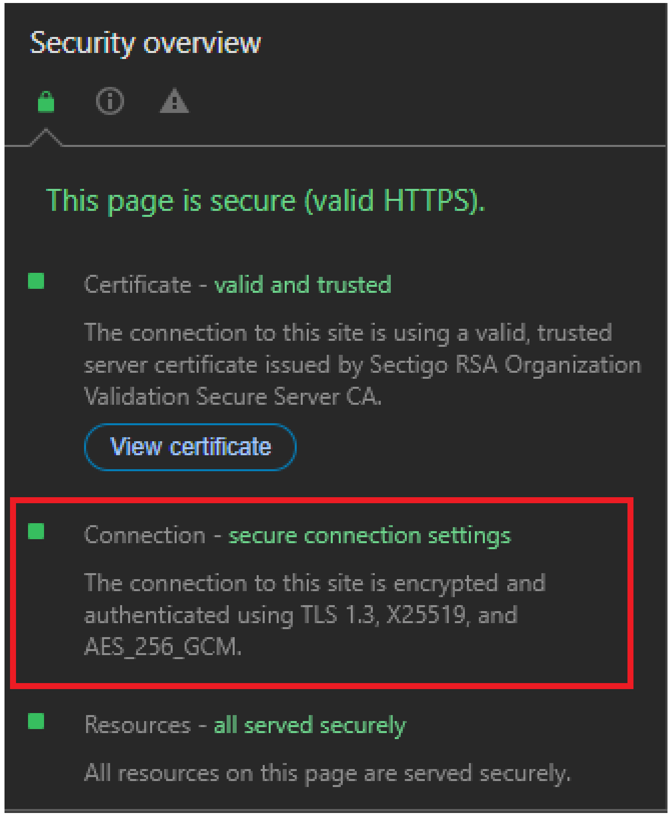
The result of Exercise 3.13 tells us that

$$\boxed{m \equiv m^{1+k\varphi(n)} \pmod{n},}$$

First decryption

https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf
encrypting both the encrypted bit stream and the first decryption algorithm with a second encryption algorithm to yield a second bit stream;	The standard practices encrypting both the encrypted bit stream (e.g., encrypted digital certificate) and the first decryption algorithm (e.g., signature decryption algorithm) with a second encryption algorithm (e.g., cipher suit selected from one of the AEAD algorithms such as TLS_AES_256_GCM_SHA384, etc.) to yield a second bit stream (e.g., TLS ciphertext bitstream). The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA algorithm) and generates a ciphertext. The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it.

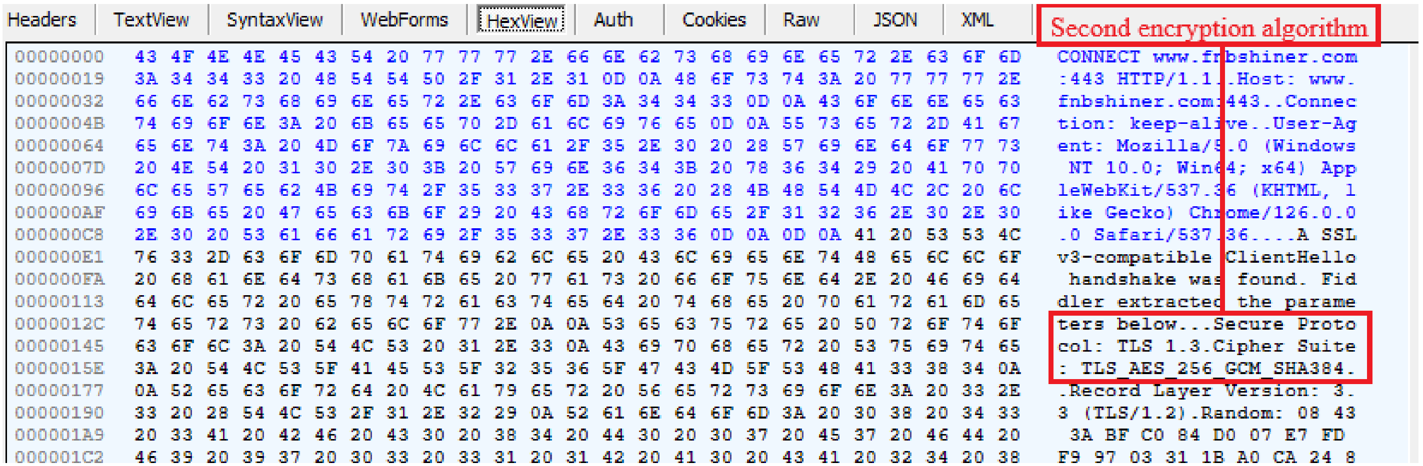
	The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate. The standard further practices encrypting the authentication message, including the encrypted digital certification and the certificate verify message, with a second decryption algorithm i.e., AEAD algorithm such as TLS_AES_256_GCM_SHA384, etc.  https://www.fnbshiner.com/

The Transport Layer Security (TLS) Protocol Version 1.3

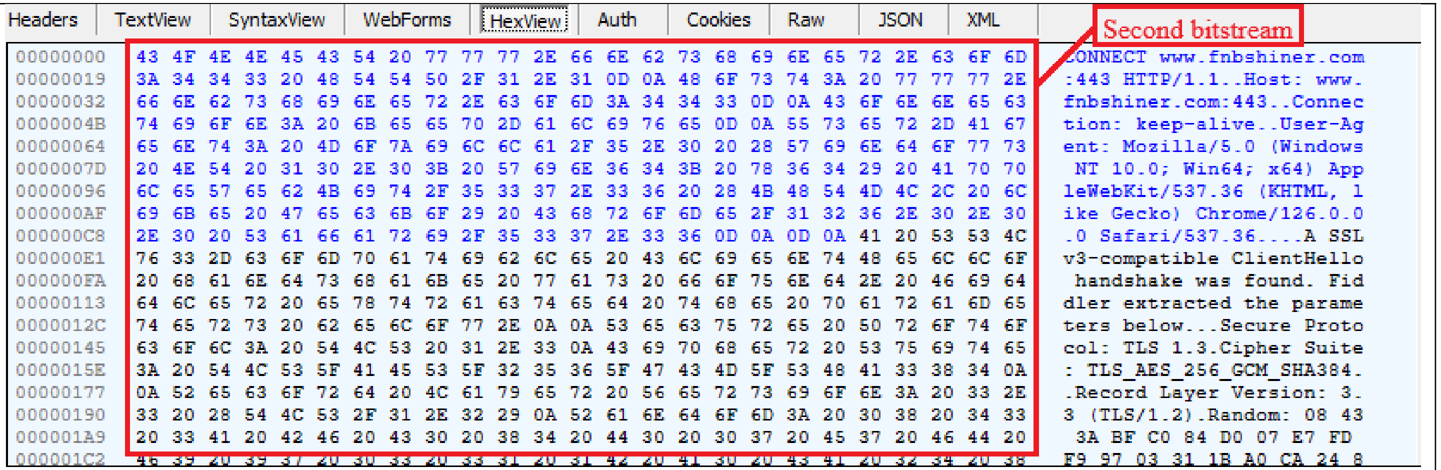
Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

https://datatracker.ietf.org/doc/html/rfc8446



Source: Fiddler Capture



Source: Fiddler Capture

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm.

As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message

	is communicated between the client and the server. **5. Record Protocol** The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients. TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see Appendix D.4). https://datatracker.ietf.org/doc/html/rfc8446#section-1 **2. Protocol Overview** Negotiating encryption algorithm The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic. https://datatracker.ietf.org/doc/html/rfc8446

	TLS consists of two primary components: - A handshake protocol (Section 4) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack. Negotiating encryption algos - A record protocol (Section 5) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys. https://datatracker.ietf.org/doc/html/rfc8446

	5.1. Record Layer The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^14 bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2. Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that: - Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **5.2. Record Payload Protection** The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see Section 5.3), and the additional data input is the record header. I.e., `additional_data = TLSCiphertext.opaque_type ||` `TLSCiphertext.legacy_record_version ||` `TLSCiphertext.length` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case. https://datatracker.ietf.org/doc/html/rfc5116

	2.1. Authenticated Encryption The authenticated encryption operation has four inputs, each of which is an octet string: A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom. A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in Section 3.2, and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces. A plaintext P, which contains the data to be encrypted and authenticated. The associated data A, which contains the data to be authenticated, but not encrypted. https://datatracker.ietf.org/doc/html/rfc5116

	The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1 Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [RFC5116], Section 4). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows: https://datatracker.ietf.org/doc/html/rfc8446#section-1

	This specification defines the following cipher suites for use with TLS 1.3. | Description | Value | | TLS_AES_128_GCM_SHA256 | {0x13,0x01} | | TLS_AES_256_GCM_SHA384 | {0x13,0x02} | | TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} | | TLS_AES_128_CCM_SHA256 | {0x13,0x04} | | TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} | https://datatracker.ietf.org/doc/html/rfc8446#section-1 All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4. Authentication Messages As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [sender]_handshake_traffic_secret. https://datatracker.ietf.org/doc/html/rfc8446#section-1

Figure 1 below shows the basic full TLS handshake:

```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

Digital Content

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.1.1. Cryptographic Negotiation In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello: - A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports. Second encryption - A "supported_groups" (Section 4.2.7) extension which indicates the (EC)DHE groups which the client supports and a "key_share" (Section 4.2.8) extension which contains (EC)DHE shares for some or all of these groups. - A "signature_algorithms" (Section 4.2.3) extension which indicates the signature algorithms which the client can accept. A "signature_algorithms_cert" extension (Section 4.2.3) may also be added to indicate certificate-specific signature algorithms. First encryption - A "pre_shared_key" (Section 4.2.11) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" (Section 4.2.9) extension which indicates the key exchange modes that may be used with PSKs. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2. Certificate This message conveys the endpoint's certificate chain to the peer. The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK). The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2.3. Client Certificate Selection The following rules apply to certificates sent by the client: - The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]). - If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs. - The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS. - If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5. https://datatracker.ietf.org/doc/html/rfc8446#section-1

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
   struct {
       SignatureScheme algorithm;
       opaque signature<0..2^16-1>;
   } CertificateVerify;
```

First decryption algorithm information

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	4.3.2. Certificate Request A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions. Structure of this message: `struct {` `opaque certificate_request_context<0..2^8-1>;` `Extension extensions<2..2^16-1>;` `} CertificateRequest;` https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages. extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions. https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	- RSASSA-PSS signature schemes are defined in Section 4.2.3. - The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello. - The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates. https://datatracker.ietf.org/doc/html/rfc8446#section-1 If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message. In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.2.3. Signature Algorithms TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2). The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension. https://datatracker.ietf.org/doc/html/rfc8446#section-1

<table>
<tr>
<td></td>
<td>
The "extension_data" field of these extensions contains a SignatureSchemeList value:
<pre>
   enum {
       /* RSASSA-PKCS1-v1_5 algorithms */
       rsa_pkcs1_sha256(0x0401),
       rsa_pkcs1_sha384(0x0501),
       rsa_pkcs1_sha512(0x0601),

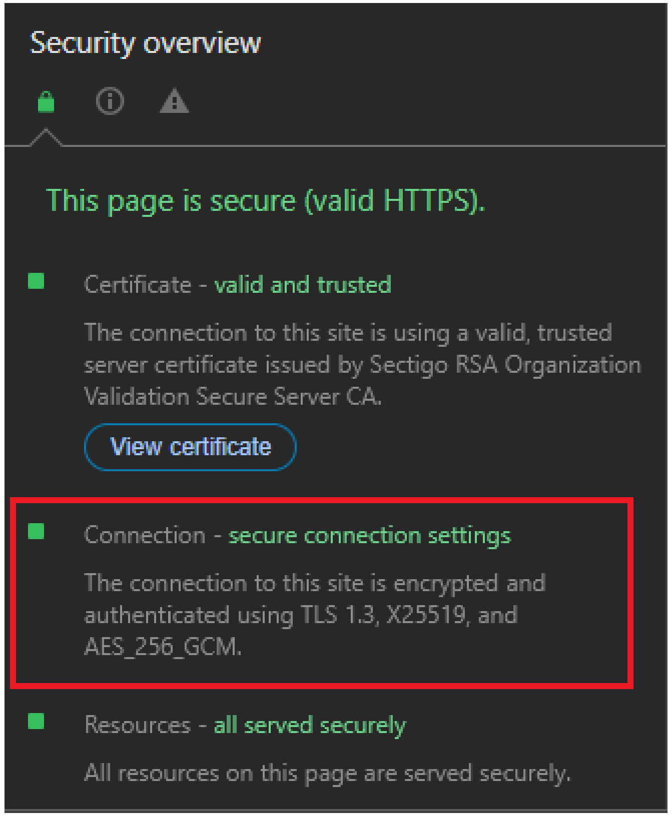
       /* ECDSA algorithms */
       ecdsa_secp256r1_sha256(0x0403),
       ecdsa_secp384r1_sha384(0x0503),
       ecdsa_secp521r1_sha512(0x0603),

       /* RSASSA-PSS algorithms with public key OID rsaEncryption */
       rsa_pss_rsae_sha256(0x0804),
       rsa_pss_rsae_sha384(0x0805),
       rsa_pss_rsae_sha512(0x0806),

       /* EdDSA algorithms */
       ed25519(0x0807),
       ed448(0x0808),

       /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */
       rsa_pss_pss_sha256(0x0809),
       rsa_pss_pss_sha384(0x080a),
       rsa_pss_pss_sha512(0x080b),
</pre>
https://datatracker.ietf.org/doc/html/rfc8446#section-1
</td>
</tr>
<tr>
<td>associating a second decryption algorithm with the second bit stream.</td>
<td>
The standard practices associating a second decryption algorithm (e.g., cipher suit selected from one of the AEAD algorithms such as TLS_AES_256_GCM_SHA384, etc.) with the second bit stream (e.g., TLS ciphertext bitstream).

The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA algorithm) and generates a ciphertext.
</td>
</tr>
</table>

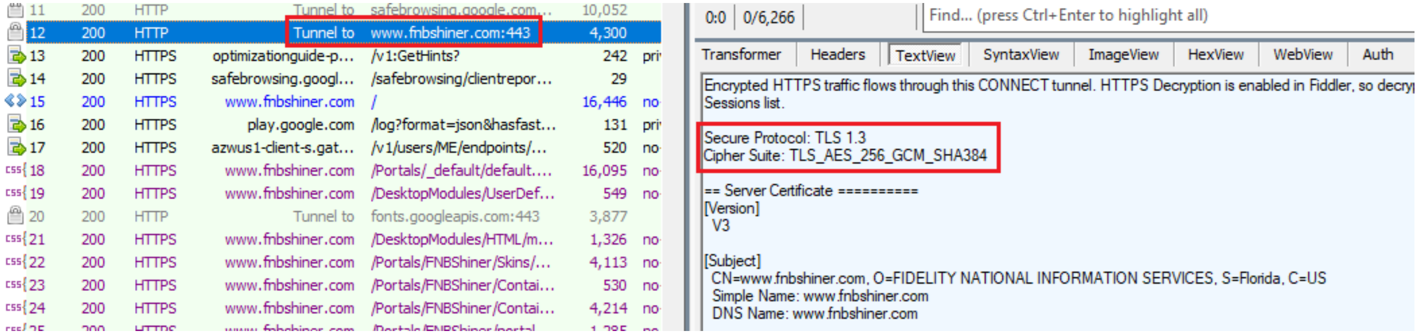
	The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it. The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate. The standard further practices encrypting the authentication message, including the encrypted digital certification and the certificate verify message, with a second decryption algorithm i.e., AEAD algorithm such as TLS_AES_256_GCM_SHA384, etc.  https://www.fnbshiner.com/

The Transport Layer Security (TLS) Protocol Version 1.3

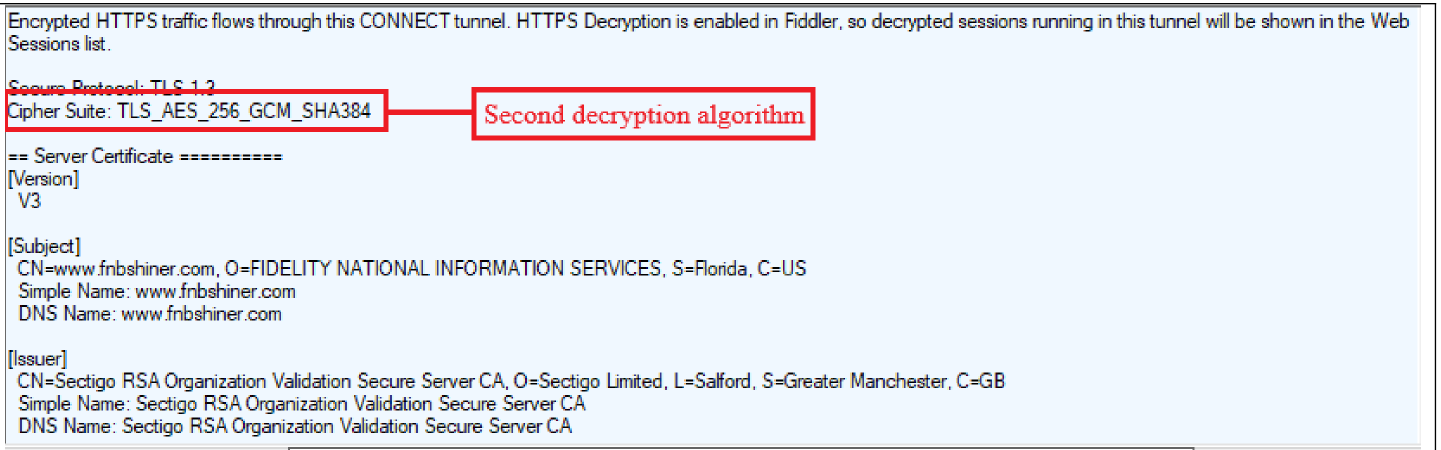
Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

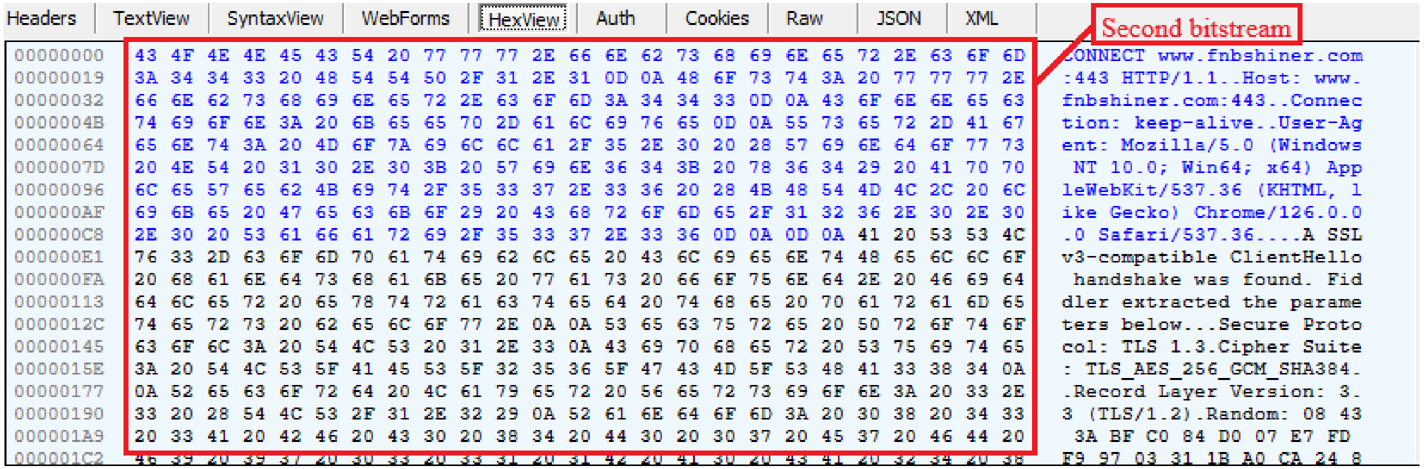
https://datatracker.ietf.org/doc/html/rfc8446



Source: Fiddler Capture



Source: Fiddler Capture



Source: Fiddler Capture

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for

	encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm. As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server. Further, the AEAD encrypted message comprises a ciphertext (e.g., encrypted ciphertext after the encryption by the second encryption algorithm), nonce (e.g., associating second decryption algo), key and associated data. The maximum length of nonce is a cipher suit specific element. The nonce and associated data are utilized in decryption of the AEAD encrypted message.

	5. Record Protocol The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients. TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see Appendix D.4). https://datatracker.ietf.org/doc/html/rfc8446#section-1 **2. Protocol Overview** Negotiating encryption algorithm The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic. https://datatracker.ietf.org/doc/html/rfc8446

	TLS consists of two primary components: - A handshake protocol (Section 4) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack. Negotiating encryption algos - A record protocol (Section 5) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys. https://datatracker.ietf.org/doc/html/rfc8446

	5.1. Record Layer The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^14 bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2. Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that: - Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **5.2. Record Payload Protection** The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see Section 5.3), and the additional data input is the record header. I.e., `additional_data = TLSCiphertext.opaque_type ||` `TLSCiphertext.legacy_record_version ||` `TLSCiphertext.length` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case. https://datatracker.ietf.org/doc/html/rfc5116

	2.1. Authenticated Encryption The authenticated encryption operation has four inputs, each of which is an octet string: A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom. A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in Section 3.2, and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces. A plaintext P, which contains the data to be encrypted and authenticated. The associated data A, which contains the data to be authenticated, but not encrypted. https://datatracker.ietf.org/doc/html/rfc5116

	2.2. Authenticated Decryption Second decryption algorithm The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure). https://datatracker.ietf.org/doc/html/rfc5116 The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [RFC5116], Section 4). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows: https://datatracker.ietf.org/doc/html/rfc8446#section-1 This specification defines the following cipher suites for use with TLS 1.3. | Description | Value | | TLS_AES_128_GCM_SHA256 | {0x13,0x01} | | TLS_AES_256_GCM_SHA384 | {0x13,0x02} | | TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} | | TLS_AES_128_CCM_SHA256 | {0x13,0x04} | | TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} | https://datatracker.ietf.org/doc/html/rfc8446#section-1

	All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **4.4. Authentication Messages** As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [sender]_handshake_traffic_secret. https://datatracker.ietf.org/doc/html/rfc8446#section-1

Figure 1 below shows the basic full TLS handshake:

```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

Digital Content

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2. Certificate This message conveys the endpoint's certificate chain to the peer. The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK). The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2.3. Client Certificate Selection The following rules apply to certificates sent by the client: - The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]). - If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs. - The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS. - If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.3. Certificate Verify This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message. Structure of this message: `struct {` `SignatureScheme algorithm;` `opaque signature<0..2^16-1>;` `} CertificateVerify;` The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely: https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	4.3.2. Certificate Request A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions. Structure of this message: `struct {` `opaque certificate_request_context<0..2^8-1>;` `Extension extensions<2..2^16-1>;` `} CertificateRequest;` https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

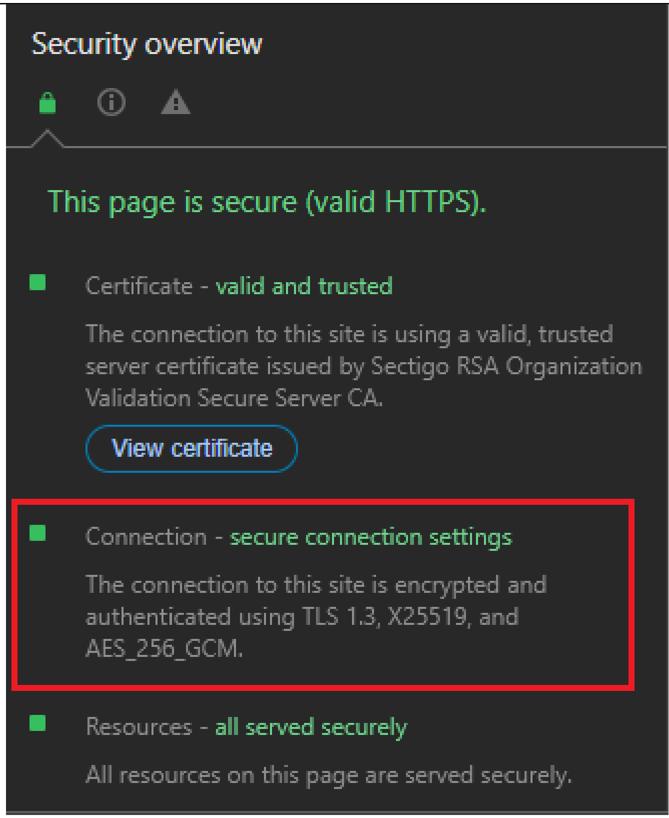
	certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages. extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions. https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	- RSASSA-PSS signature schemes are defined in Section 4.2.3. - The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello. - The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates. https://datatracker.ietf.org/doc/html/rfc8446#section-1 If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message. In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.2.3. Signature Algorithms TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2). The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	The "extension_data" field of these extensions contains a SignatureSchemeList value: `enum {` `    /* RSASSA-PKCS1-v1_5 algorithms */` `    rsa_pkcs1_sha256(0x0401),` `    rsa_pkcs1_sha384(0x0501),` `    rsa_pkcs1_sha512(0x0601),` `    /* ECDSA algorithms */` `    ecdsa_secp256r1_sha256(0x0403),` `    ecdsa_secp384r1_sha384(0x0503),` `    ecdsa_secp521r1_sha512(0x0603),` `    /* RSASSA-PSS algorithms with public key OID rsaEncryption */` `    rsa_pss_rsae_sha256(0x0804),` `    rsa_pss_rsae_sha384(0x0805),` `    rsa_pss_rsae_sha512(0x0806),` `    /* EdDSA algorithms */` `    ed25519(0x0807),` `    ed448(0x0808),` `    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */` `    rsa_pss_pss_sha256(0x0809),` `    rsa_pss_pss_sha384(0x080a),` `    rsa_pss_pss_sha512(0x080b),` https://datatracker.ietf.org/doc/html/rfc8446#section-1
20. The system of claim 19, further operable for decrypting the first bit stream and the second	The standard further discloses decrypting the first bit stream (e.g., encrypted digital certificate with signature encryption algorithm i.e., SHA-256 RSA, etc.) and the second bit stream (e.g., a second-level encryption with AEAD encryption algorithm such as TLS_AES_256_GCM_SHA384, etc.) with the first associated decryption

bit stream with the first associated decryption algorithm and the second associated decryption algorithm wherein the decryption is accomplished by a target unit.	algorithm (e.g., signature decryption algorithm i.e., SHA-256 RSA, etc.) and the second associated decryption algorithm (e.g., cipher suit selected from one of the AEAD decryption algorithms such as TLS_AES_256_GCM_SHA384, etc.) wherein the decryption is accomplished by a target unit (e.g., a server of the accused instrumentality). The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA algorithm) and generates a ciphertext. The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it. The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate. The standard further practices encrypting the authentication message, including the encrypted digital certification and the certificate verify message, with a second decryption algorithm i.e., AEAD algorithm such as TLS_AES_256_GCM_SHA384, etc.

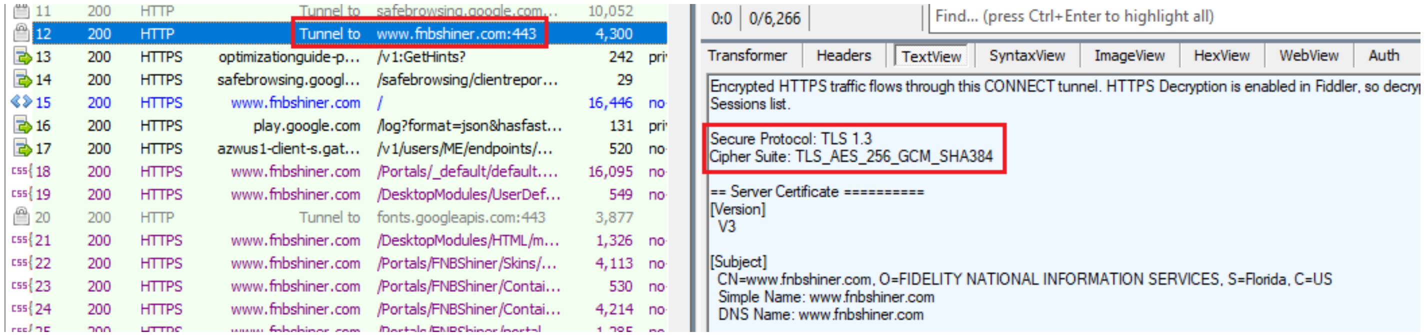
	 https://www.fnbshiner.com/

The Transport Layer Security (TLS) Protocol Version 1.3

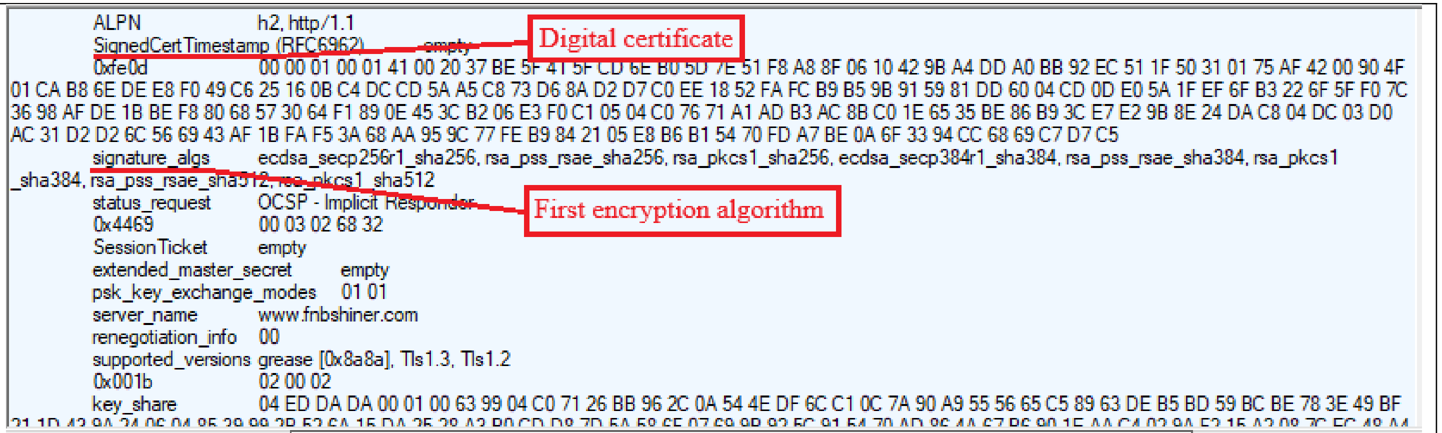
Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

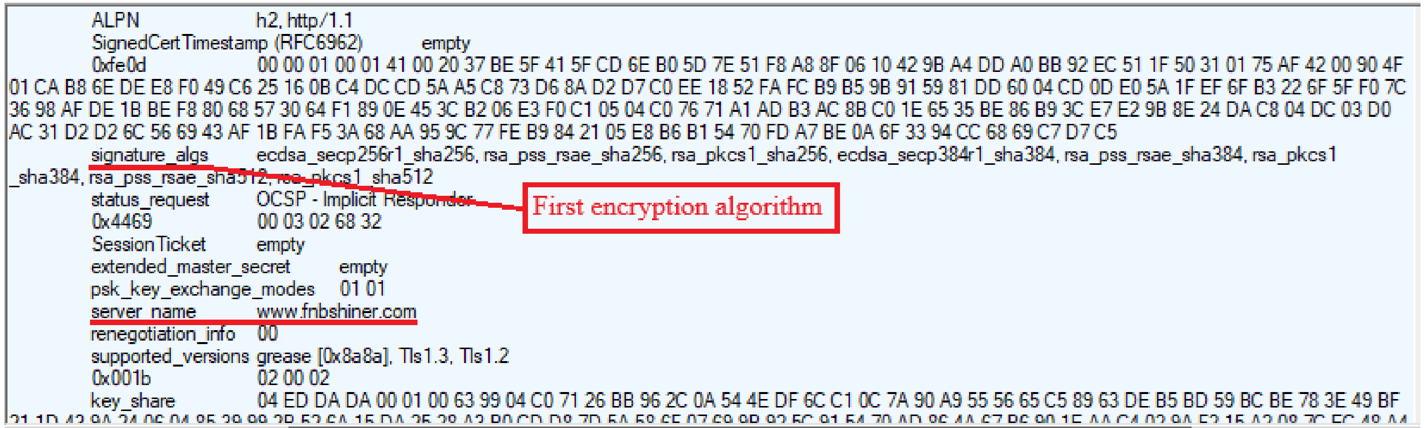
https://datatracker.ietf.org/doc/html/rfc8446



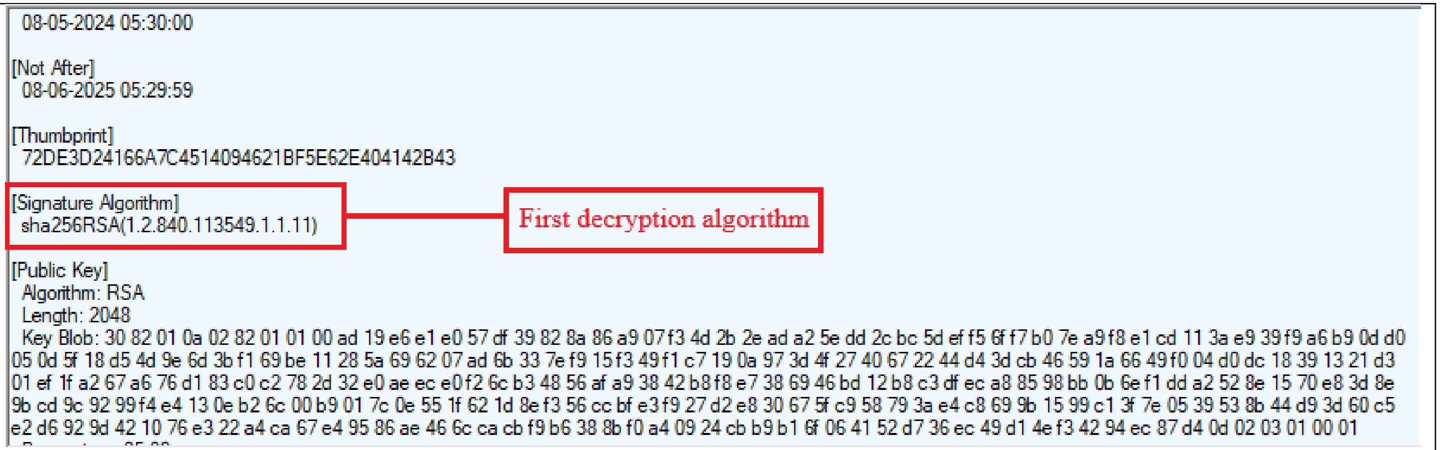
Source: Fiddler Capture



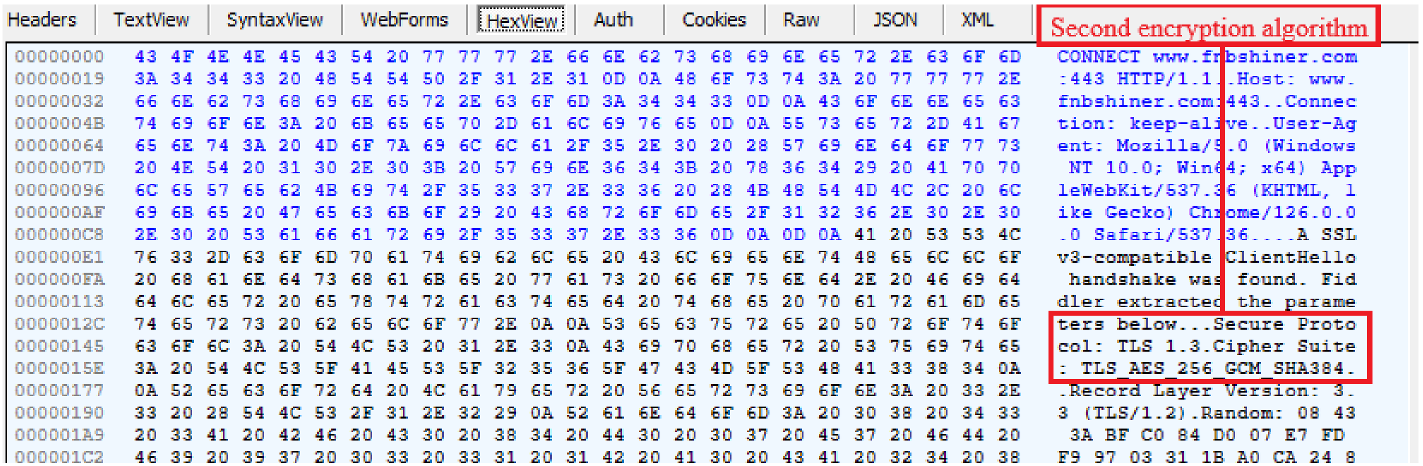
Source: Fiddler Capture



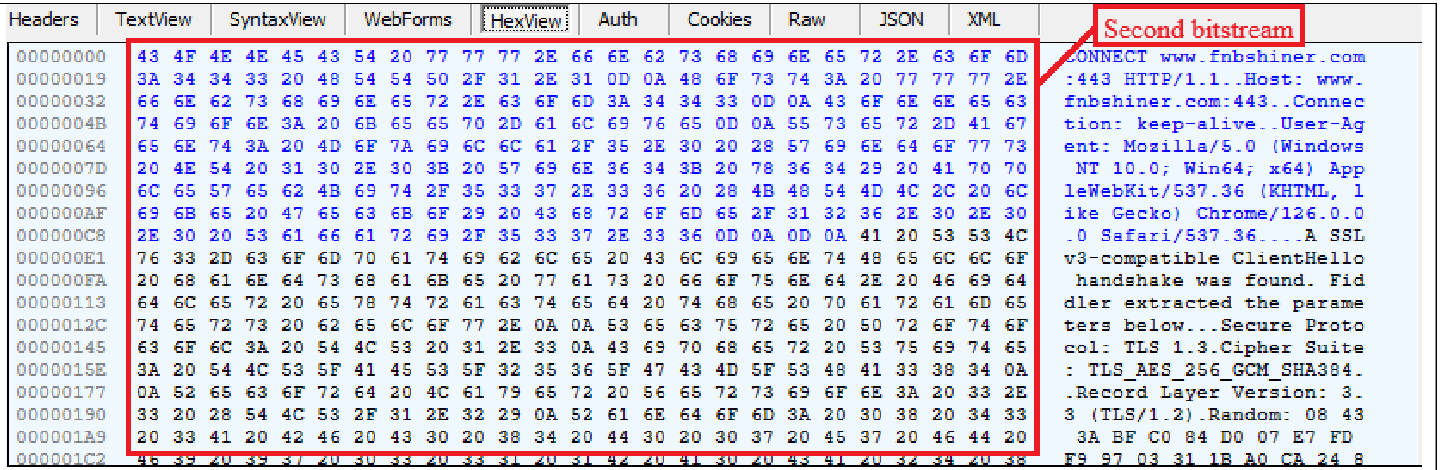
Source: Fiddler Capture



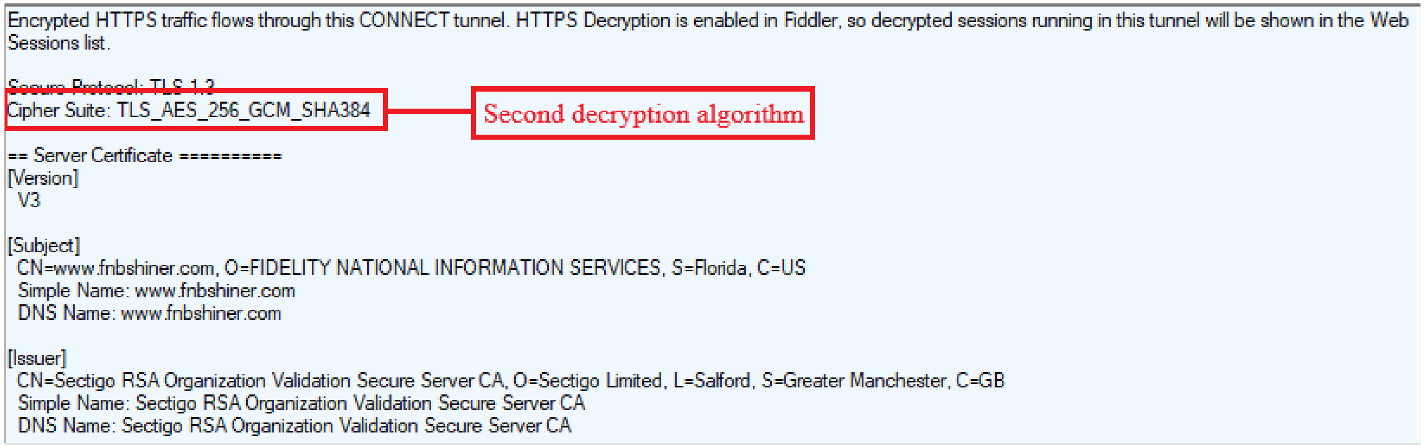
Source: Fiddler Capture



Source: Fiddler Capture



Source: Fiddler Capture



Source: Fiddler Capture

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for

	encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm. As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server. Further, the AEAD encrypted message comprises a ciphertext (e.g., encrypted ciphertext after the encryption by the second encryption algorithm), nonce (e.g., associating second decryption algo), key and associated data. The maximum length of nonce is a cipher suit specific element. The nonce and associated data are utilized in decryption of the AEAD encrypted message.

	5. Record Protocol The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients. TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see Appendix D.4). https://datatracker.ietf.org/doc/html/rfc8446#section-1 **2. Protocol Overview** Negotiating encryption algorithm The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic. https://datatracker.ietf.org/doc/html/rfc8446

	TLS consists of two primary components: - A handshake protocol (Section 4) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack. Negotiating encryption algos - A record protocol (Section 5) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys. https://datatracker.ietf.org/doc/html/rfc8446

	5.1. Record Layer The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^14 bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2. Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that: - Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **5.2. Record Payload Protection** The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see Section 5.3), and the additional data input is the record header. I.e., additional_data = TLSCiphertext.opaque_type || TLSCiphertext.legacy_record_version || TLSCiphertext.length https://datatracker.ietf.org/doc/html/rfc8446#section-1

	The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case. https://datatracker.ietf.org/doc/html/rfc5116

	2.1. Authenticated Encryption The authenticated encryption operation has four inputs, each of which is an octet string: A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom. A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in Section 3.2, and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces. A plaintext P, which contains the data to be encrypted and authenticated. The associated data A, which contains the data to be authenticated, but not encrypted. https://datatracker.ietf.org/doc/html/rfc5116

	2.2. Authenticated Decryption Second decryption algorithm The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure). https://datatracker.ietf.org/doc/html/rfc5116 The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [RFC5116], Section 4). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows: https://datatracker.ietf.org/doc/html/rfc8446#section-1 This specification defines the following cipher suites for use with TLS 1.3. | Description | Value | | TLS_AES_128_GCM_SHA256 | {0x13,0x01} | | TLS_AES_256_GCM_SHA384 | {0x13,0x02} | | TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} | | TLS_AES_128_CCM_SHA256 | {0x13,0x04} | | TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} | https://datatracker.ietf.org/doc/html/rfc8446#section-1

	All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **4.4. Authentication Messages** As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [sender]_handshake_traffic_secret. https://datatracker.ietf.org/doc/html/rfc8446#section-1

Figure 1 below shows the basic full TLS handshake:

```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

Digital Content

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2. Certificate This message conveys the endpoint's certificate chain to the peer. The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK). The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2.3. Client Certificate Selection The following rules apply to certificates sent by the client: - The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]). - If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs. - The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS. - If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.3. Certificate Verify This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message. Structure of this message: `struct {` `SignatureScheme algorithm;` `opaque signature<0..2^16-1>;` `} CertificateVerify;` The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely: https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	4.3.2. Certificate Request A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions. Structure of this message: `struct {` `opaque certificate_request_context<0..2^8-1>;` `Extension extensions<2..2^16-1>;` `} CertificateRequest;` https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages. extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions. https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

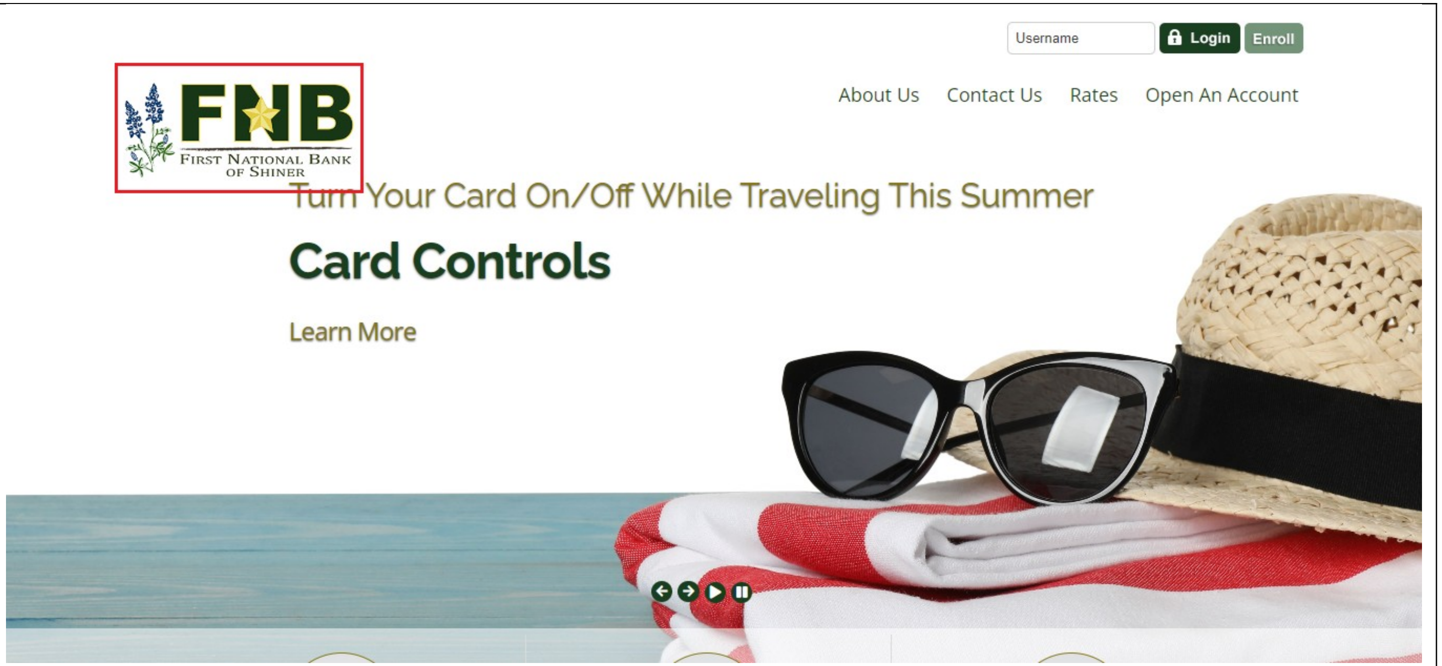
	- RSASSA-PSS signature schemes are defined in Section 4.2.3. - The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello. - The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates. https://datatracker.ietf.org/doc/html/rfc8446#section-1 If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message. In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages. https://datatracker.ietf.org/doc/html/rfc8446#section-1

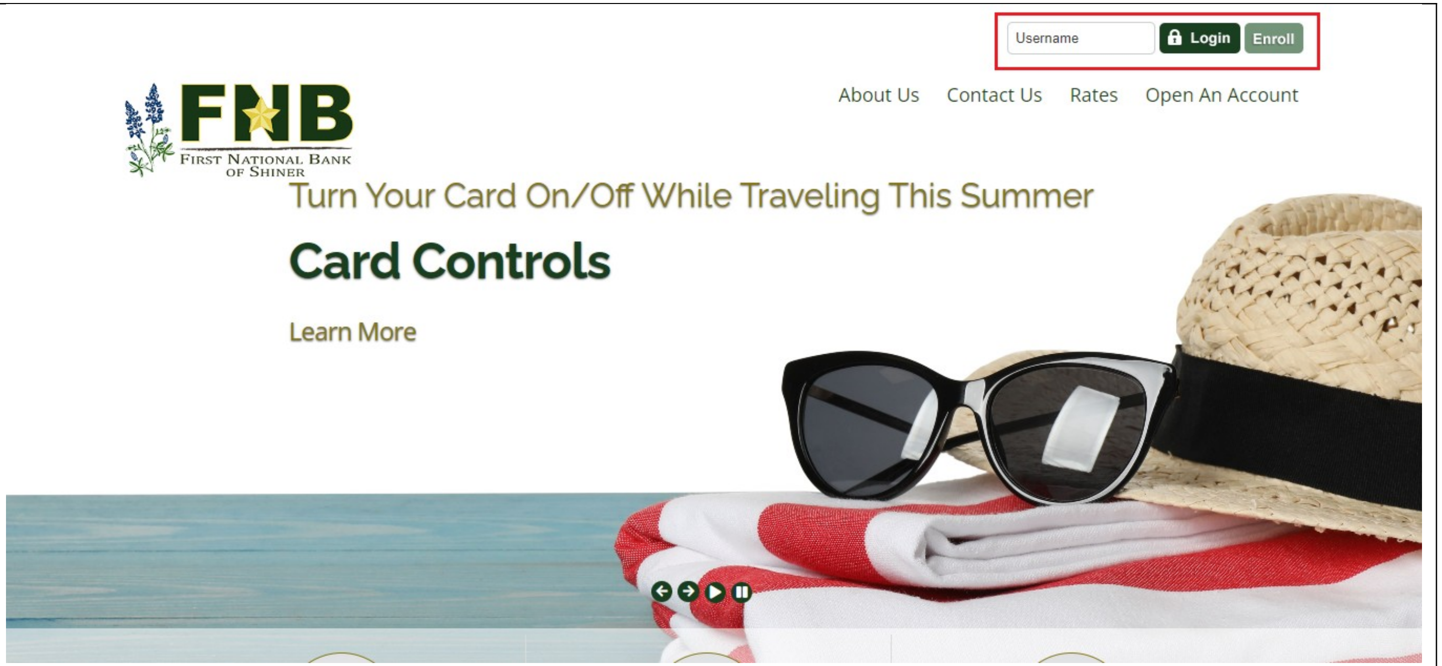
	4.2.3. Signature Algorithms TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2). The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension. https://datatracker.ietf.org/doc/html/rfc8446#section-1

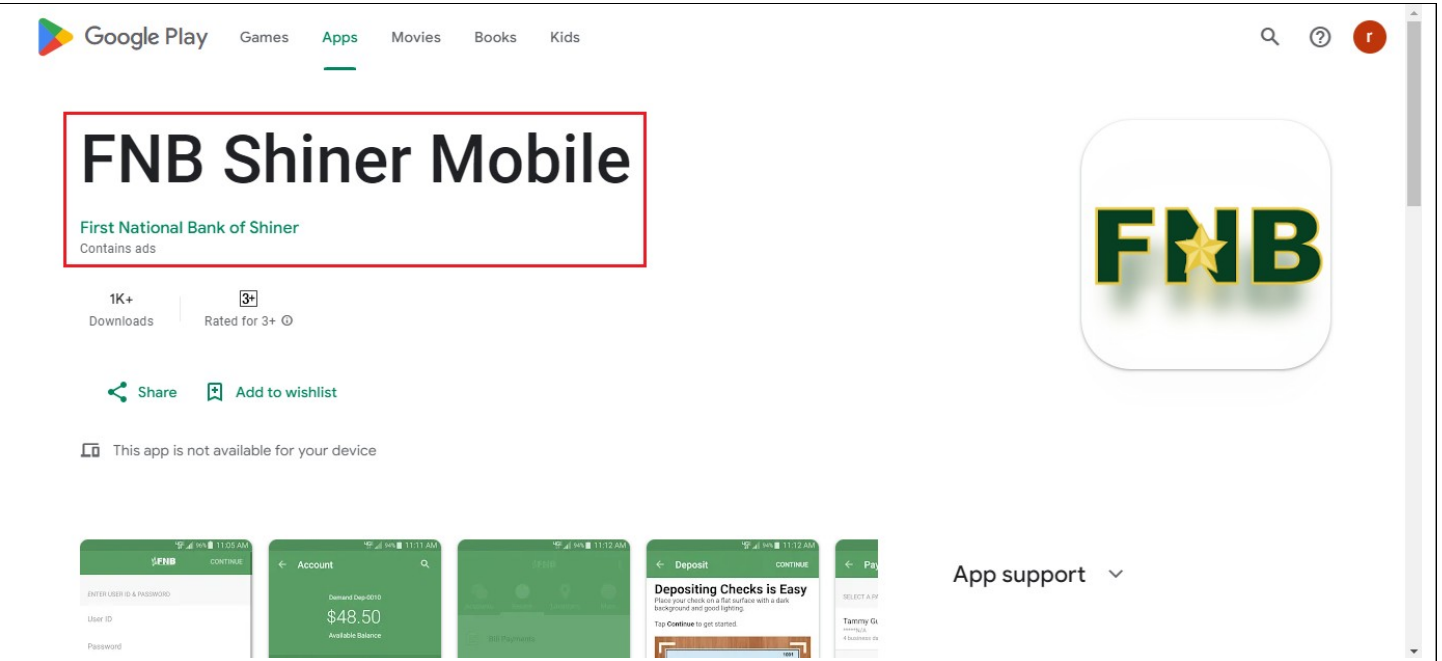
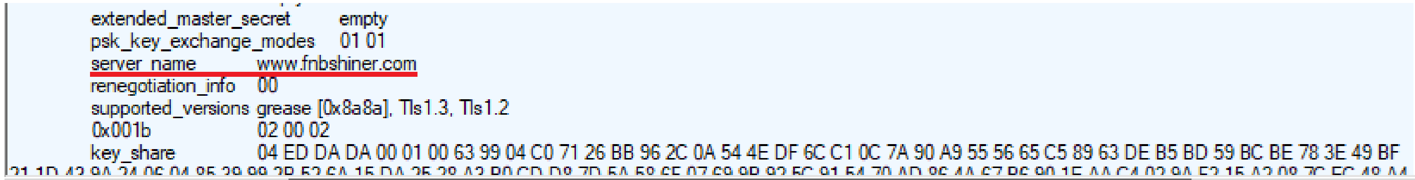
	The "extension_data" field of these extensions contains a SignatureSchemeList value: `enum {` `    /* RSASSA-PKCS1-v1_5 algorithms */` `    rsa_pkcs1_sha256(0x0401),` `    rsa_pkcs1_sha384(0x0501),` `    rsa_pkcs1_sha512(0x0601),` `    /* ECDSA algorithms */` `    ecdsa_secp256r1_sha256(0x0403),` `    ecdsa_secp384r1_sha384(0x0503),` `    ecdsa_secp521r1_sha512(0x0603),` `    /* RSASSA-PSS algorithms with public key OID rsaEncryption */` `    rsa_pss_rsae_sha256(0x0804),` `    rsa_pss_rsae_sha384(0x0805),` `    rsa_pss_rsae_sha512(0x0806),` `    /* EdDSA algorithms */` `    ed25519(0x0807),` `    ed448(0x0808),` `    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */` `    rsa_pss_pss_sha256(0x0809),` `    rsa_pss_pss_sha384(0x080a),` `    rsa_pss_pss_sha512(0x080b),` https://datatracker.ietf.org/doc/html/rfc8446#section-1 As shown below, the receiving party will be able to decrypt the encrypted message with the provided signature decryption algorithm information i.e., SHA-256 RSA decryption algorithm.

	We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q, compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de = 1 \pmod{\varphi(n)}$. we know that $de-1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$. Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is $\boxed{c = m^e \bmod n.}$ First encryption To decrypt, we compute c^d mod n to obtain $c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$ The result of Exercise 3.13 tells us that $\boxed{m \equiv m^{1+k\varphi(n)} \pmod{n},}$ First decryption https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf
21. The system of claim 20, wherein the decrypting is done using a key associated with each decryption algorithm.	The standard practices the method such that the decrypting is done using a key (e.g., decryption key) associated with each decryption algorithm (e.g., signature decryption algorithm such as SHA-256RSA, etc., and AEAD decryption algorithm such as TLS_AES_256_GCM_SHA384, etc.).

	 https://www.fnbshiner.com/

	 https://www.fnbshiner.com/

	 https://play.google.com/store/apps/details?id=com.mfoundry.mb.android.mb_113106833&hl=en_US  extended_master_secret empty psk_key_exchange_modes 01 01 server_name www.fnbshiner.com renegotiation_info 00 supported_versions grease [0x8a8a], Tls1.3, Tls1.2 0x001b 02 00 02 key_share 04 ED DA DA 00 01 00 63 99 04 C0 71 26 BB 96 2C 0A 54 4E DF 6C C1 0C 7A 90 A9 55 56 65 C5 89 63 DE B5 BD 59 BC BE 78 3E 49 BF *Source: Fiddler Capture* As shown below, the signature decryption algorithm utilizes a private key for a first decryption and the AEAD decryption algorithm uses a key K. Both the decryption techniques are decrypting using their respective associated keys.

	The "extension_data" field of these extensions contains a SignatureSchemeList value: <pre> enum { /* RSASSA-PKCS1-v1_5 algorithms */ rsa_pkcs1_sha256(0x0401), rsa_pkcs1_sha384(0x0501), rsa_pkcs1_sha512(0x0601), /* ECDSA algorithms */ ecdsa_secp256r1_sha256(0x0403), ecdsa_secp384r1_sha384(0x0503), ecdsa_secp521r1_sha512(0x0603), /* RSASSA-PSS algorithms with public key OID rsaEncryption */ rsa_pss_rsae_sha256(0x0804), rsa_pss_rsae_sha384(0x0805), rsa_pss_rsae_sha512(0x0806), /* EdDSA algorithms */ ed25519(0x0807), ed448(0x0808), /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */ rsa_pss_pss_sha256(0x0809), rsa_pss_pss_sha384(0x080a), rsa_pss_pss_sha512(0x080b),</pre>https://datatracker.ietf.org/doc/html/rfc8446#section-1

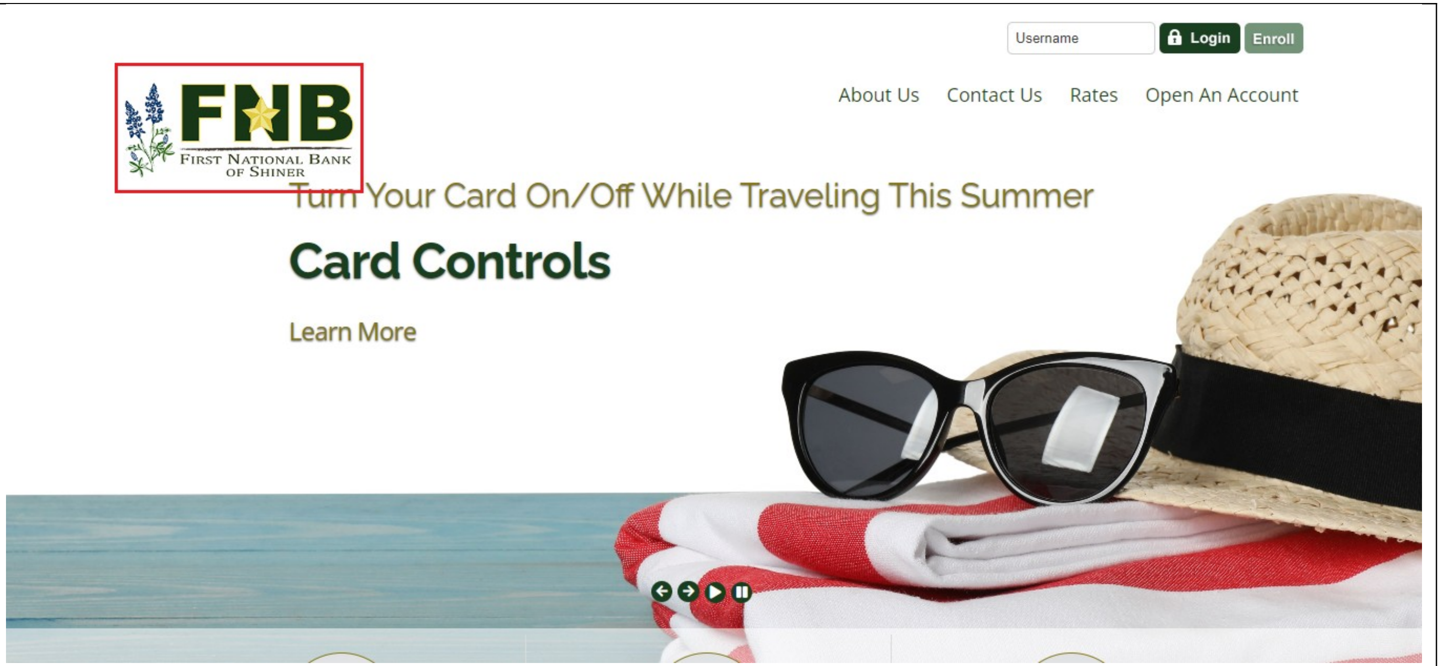
	There is also a decryption function D that takes a ciphertext and a decryption key K_D, and reproduces the plaintext message. $D(C, K_D) = P$ In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster. In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B's encryption key, encrypt a message with it, and send the result to person B. Only someone with B's decryption key, namely only B, can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

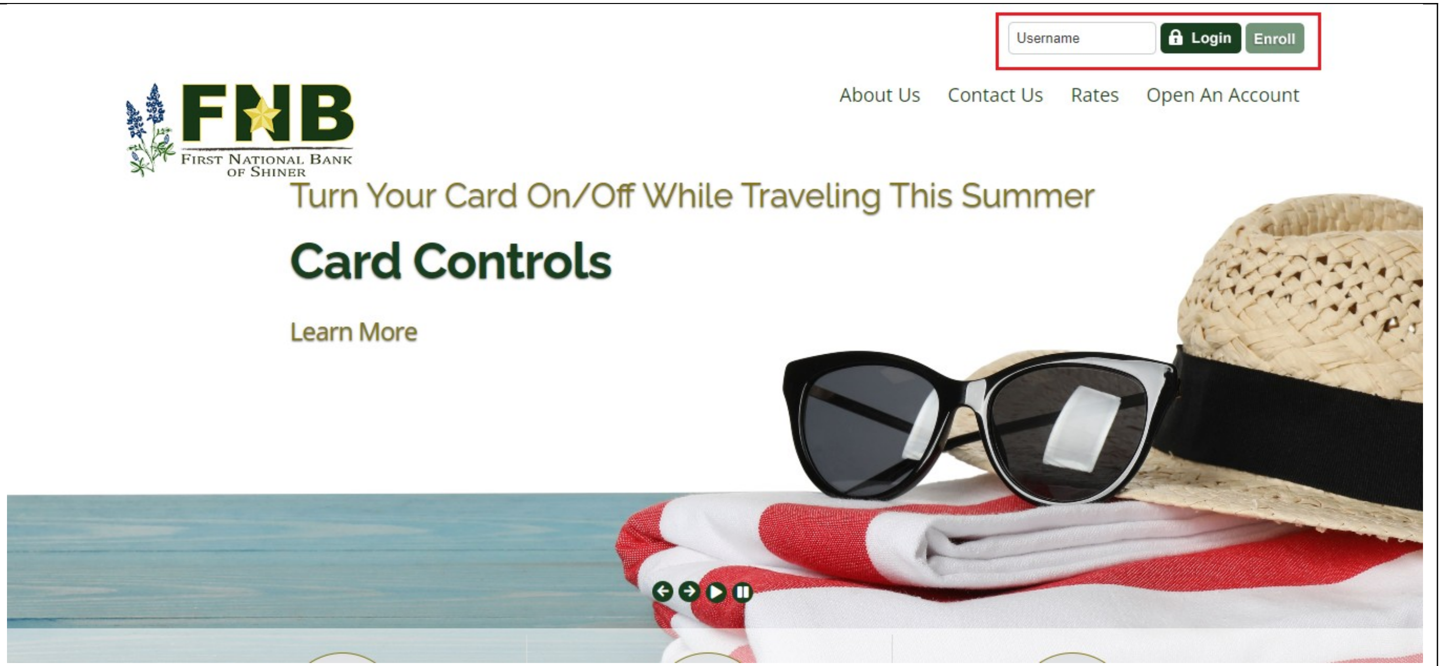
	We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q, compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de = 1 \pmod{\varphi(n)}$. we know that $de-1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$. Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is $\boxed{c = m^e \bmod n.}$ First encryption To decrypt, we compute $c^d \bmod n$ to obtain $c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$ The result of Exercise 3.13 tells us that $\boxed{m \equiv m^{1+k\varphi(n)} \pmod{n},}$ First decryption https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

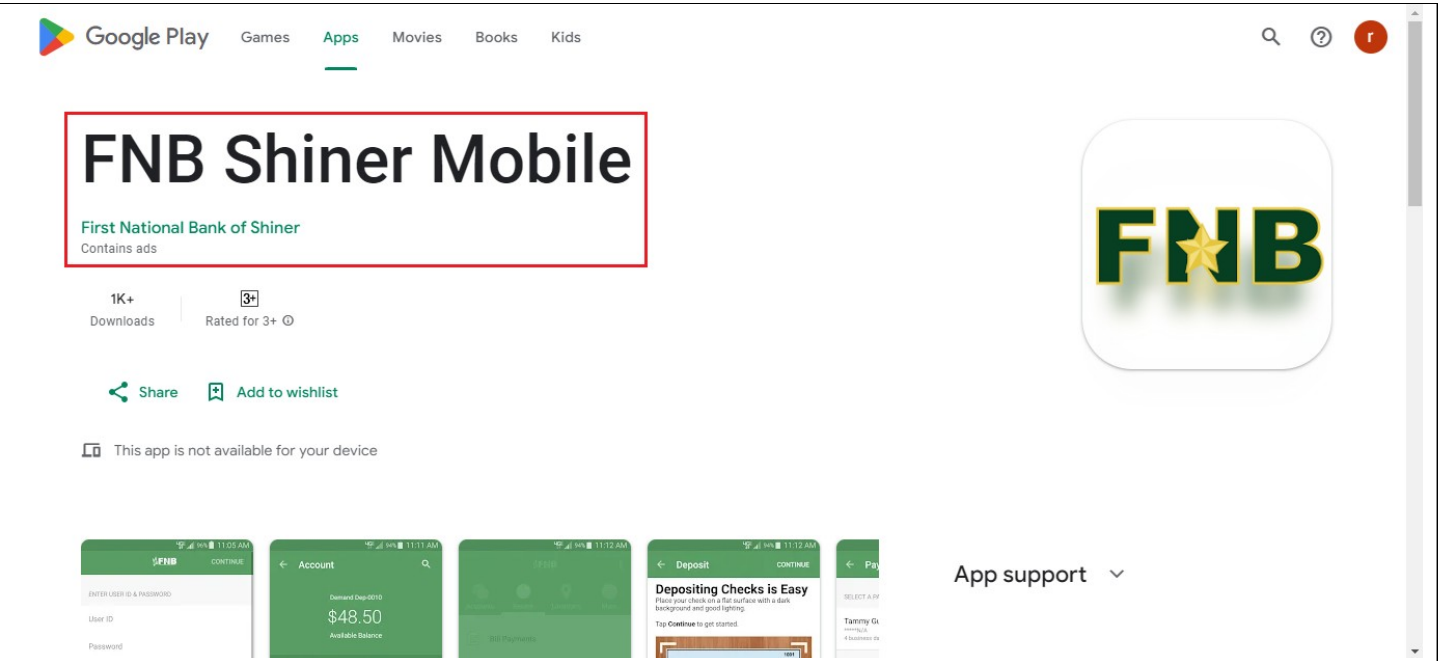
	The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	This specification defines the following cipher suites for use with TLS 1.3. ``` +------------------------------+-------------+ | Description | Value | +------------------------------+-------------+ | TLS_AES_128_GCM_SHA256 | {0x13,0x01} | | | | | TLS_AES_256_GCM_SHA384 | {0x13,0x02} | | | | | TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} | | | | | TLS_AES_128_CCM_SHA256 | {0x13,0x04} | | | | | TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} | +------------------------------+-------------+ ``` https://datatracker.ietf.org/doc/html/rfc8446#section-1

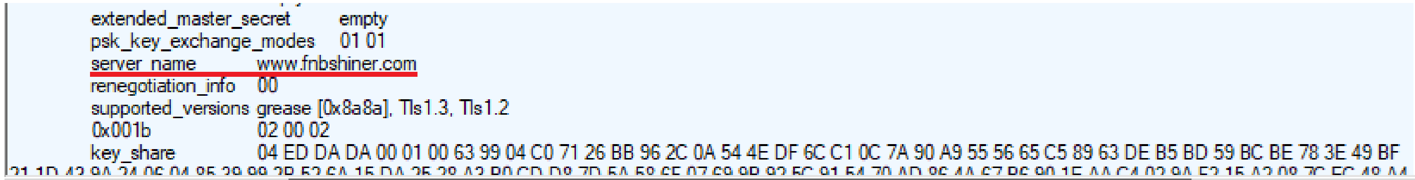
	2.2. Authenticated Decryption The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure). https://datatracker.ietf.org/doc/html/rfc5116 The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1
22. The system of claim 21, wherein the key is resident in hardware of the target unit or the key is retrieved from a server.	The standard utilized by the accused instrumentality practices the method such that the key is resident in hardware (e.g., stored in a memory storage of the server such as a database, RAM, etc.) of the target unit (e.g., server of the accused instrumentality) or the key is retrieved from a server.

	 https://www.fnbshiner.com/

	 https://www.fnbshiner.com/



https://play.google.com/store/apps/details?id=com.mfoundry.mb.android.mb_113106833&hl=en_US



Source: Fiddler Capture

	Tech Accelerator **Server hardware guide: Architecture, products and management** **3. Random access memory** RAM is the main type of memory in a computing system. RAM holds the software instructions and data needed by the processor, along with any output from the processor, such as data to be moved to a storage device. Thus, RAM works very closely with the processor and must match the processor's incredible speed and performance. This kind of fast memory is usually termed dynamic RAM, and several DRAM variations are available for servers. https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms

Tech Accelerator

Server hardware guide: Architecture, products and management

4. Hard disk drive

This hardware is responsible for reading, writing and positioning of the hard disk, which is one technology for data storage on server hardware. Developed at IBM in 1953, the [hard disk drive (HDD)]() has evolved over time from the size of a refrigerator to the standard 2.5-inch and 3.5-inch form factors.

https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms

As shown below, the server comprises a memory storage to store messages for establishing secure TLS communication. the standard discloses multiple signature encryption algorithms for a first encryption and multiple AEAD encryption algorithms for the second encryption. A signature decryption algorithm utilizes a private key for decrypting the first bitstream encrypted with the signature encryption and an AEAD decryption algorithm uses a key K for decrypting the second bitstream encrypted with the AEAD encryption. Both the decryption techniques are decrypting using their respective associated keys. A server must have a storage to store information pertaining to these algorithms and their corresponding keys such as private key, Key K, etc., to establish secure TLS communication with a client.

	Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT. https://datatracker.ietf.org/doc/html/rfc8446#

	The "extension_data" field of these extensions contains a SignatureSchemeList value: ``` enum { /* RSASSA-PKCS1-v1_5 algorithms */ rsa_pkcs1_sha256(0x0401), rsa_pkcs1_sha384(0x0501), rsa_pkcs1_sha512(0x0601), /* ECDSA algorithms */ ecdsa_secp256r1_sha256(0x0403), ecdsa_secp384r1_sha384(0x0503), ecdsa_secp521r1_sha512(0x0603), /* RSASSA-PSS algorithms with public key OID rsaEncryption */ rsa_pss_rsae_sha256(0x0804), rsa_pss_rsae_sha384(0x0805), rsa_pss_rsae_sha512(0x0806), /* EdDSA algorithms */ ed25519(0x0807), ed448(0x0808), /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */ rsa_pss_pss_sha256(0x0809), rsa_pss_pss_sha384(0x080a), rsa_pss_pss_sha512(0x080b), ``` https://datatracker.ietf.org/doc/html/rfc8446#section-1

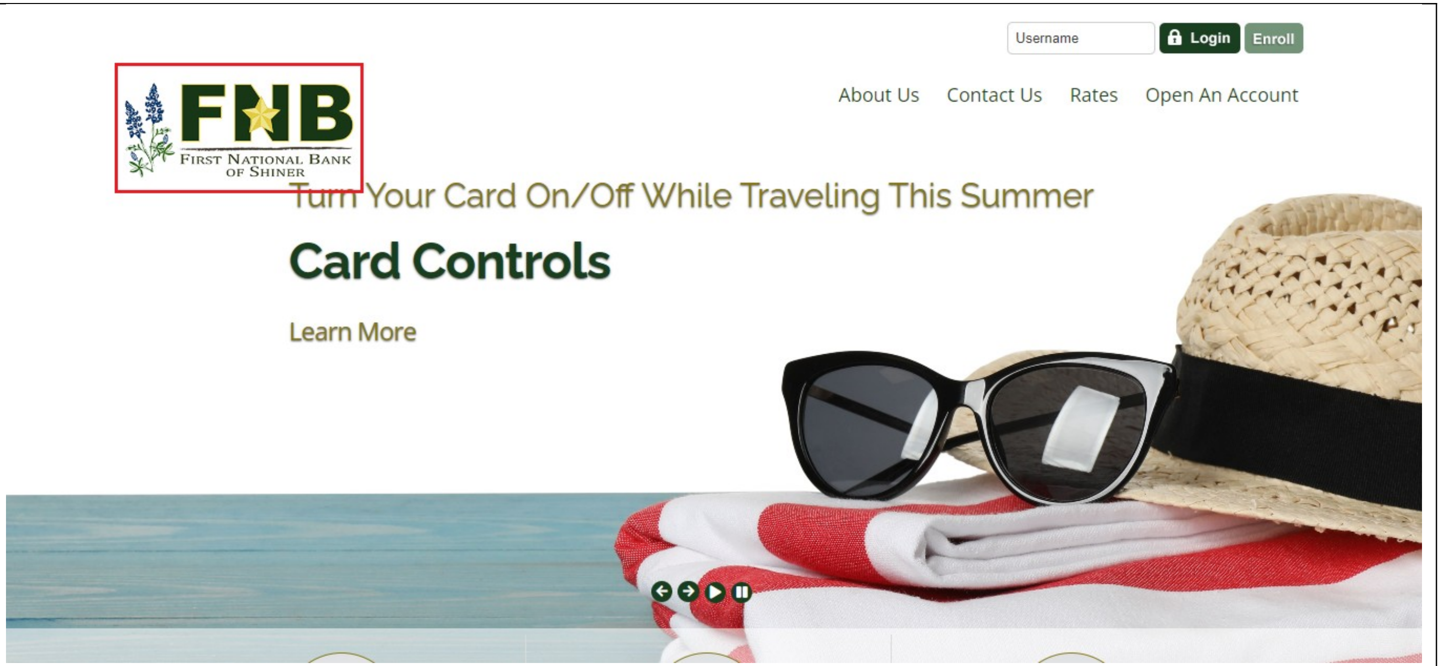
	There is also a decryption function D that takes a ciphertext and a decryption key K_D, and reproduces the plaintext message. $D(C, K_D) = P$ In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster. In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B's encryption key, encrypt a message with it, and send the result to person B. Only someone with B's decryption key, namely only B, can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

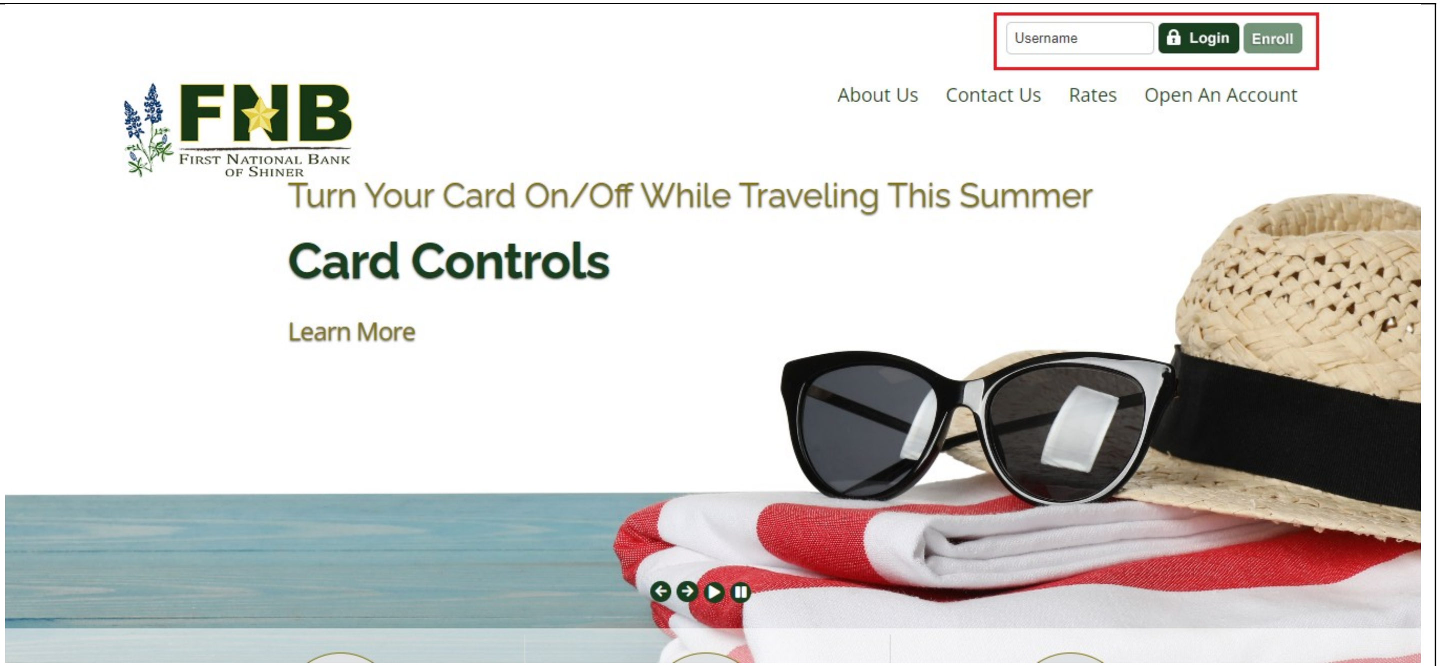
	We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q, compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de = 1 \pmod{\varphi(n)}$. we know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$. Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is $\boxed{c = m^e \bmod n.}$ First encryption To decrypt, we compute $c^d \bmod n$ to obtain $c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$ The result of Exercise 3.13 tells us that $\boxed{m \equiv m^{1+k\varphi(n)} \pmod{n},}$ First decryption https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

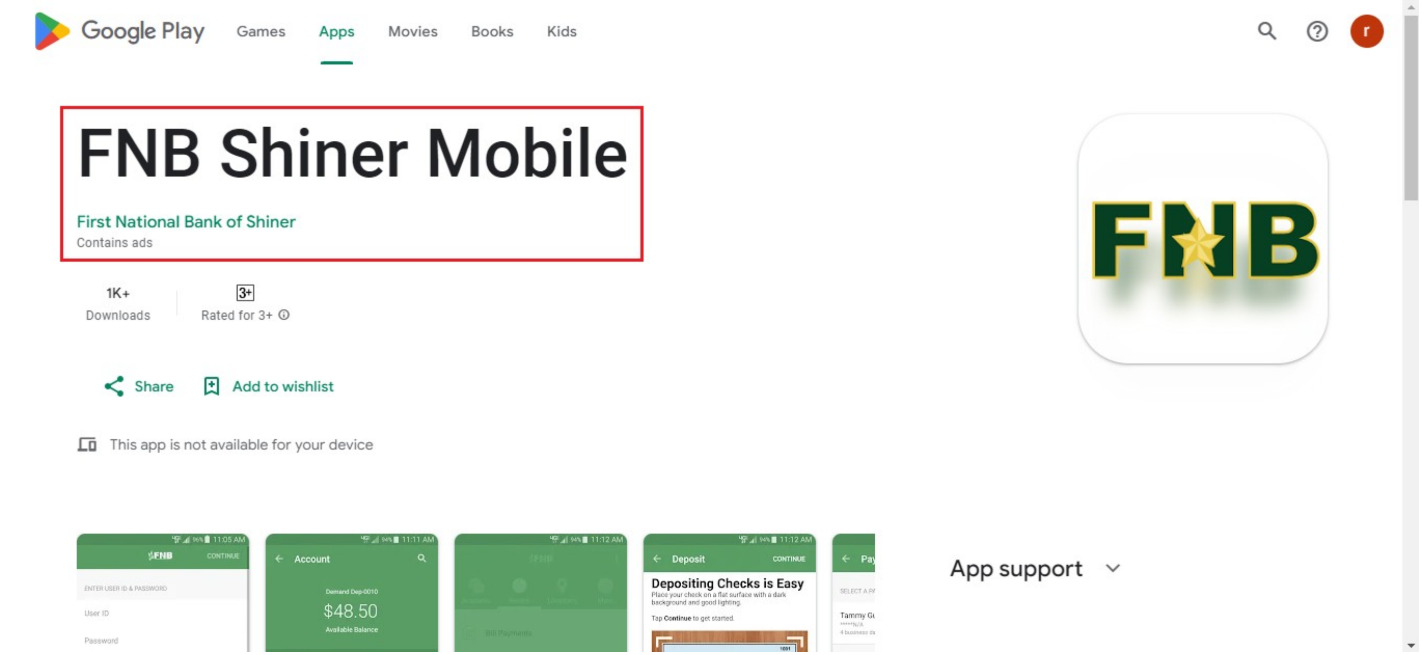
	The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions `MD5`, `SHA-1`, and `SHA-256`. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	This specification defines the following cipher suites for use with TLS 1.3. ``` +------------------------------+-------------+ | Description	Value | +------------------------------+-------------+ | TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05} | +------------------------------+-------------+ ``` https://datatracker.ietf.org/doc/html/rfc8446#section-1

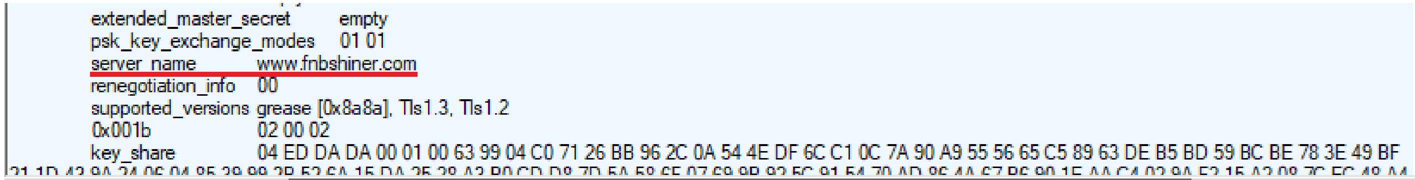
	2.2. Authenticated Decryption The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure). https://datatracker.ietf.org/doc/html/rfc5116 The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1
23. The system of claim 22, wherein the key is contained in a key data structure.	The standard utilized by the accused instrumentality practices the method such that the key (e.g., private key, Key K, etc.) is contained in a key data structure (e.g., data structure).

	 https://www.fnbshiner.com/

	 https://www.fnbshiner.com/



https://play.google.com/store/apps/details?id=com.mfoundry.mb.android.mb_113106833&hl=en_US



Source: Fiddler Capture

The accused instrumentality utilizes a server to establish a secure TLS communication with a client. The server must comprise a memory storage and store data according to a data structure to implement the standard efficiently.

Tech Accelerator

Server hardware guide: Architecture, products and management

3. Random access memory

RAM is the main type of memory in a computing system. RAM holds the software instructions and data needed by the processor, along with any output from the processor, such as data to be moved to a storage device. Thus, RAM works very closely with the processor and must match the processor's incredible speed and performance. This kind of fast memory is usually termed dynamic RAM, and several DRAM variations are available for servers.

https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms

	Tech Accelerator **Server hardware guide: Architecture, products and management** **4. Hard disk drive** This hardware is responsible for reading, writing and positioning of the hard disk, which is one technology for data storage on server hardware. Developed at IBM in 1953, the hard disk drive (HDD) has evolved over time from the size of a refrigerator to the standard 2.5-inch and 3.5-inch form factors. https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms

	A data structure is a specialized format for organizing, processing, retrieving and storing data. There are several basic and advanced types of data structures, all designed to arrange data to suit a specific purpose. Data structures make it easy for users to access and work with the data they need in appropriate ways. Most importantly, data structures frame the organization of information so that machines and humans can better understand it. In computer science and computer programming, a data structure may be selected or designed to store data for the purpose of using it with various algorithms. In some cases, the algorithm's basic operations are tightly coupled to the data structure's design. Each data structure contains information about the data values, relationships between the data and -- in some cases -- functions that can be applied to the data. https://www.techtarget.com/searchdatamanagement/definition/data-structure As shown below, the server comprises a memory storage to store messages for establishing secure TLS communication. the standard discloses multiple signature encryption algorithms for a first encryption and multiple AEAD encryption algorithms for the second encryption. A signature decryption algorithm utilizes a private key for decrypting the first bitstream encrypted with the signature encryption and an AEAD decryption algorithm uses a key K for decrypting the second bitstream encrypted with the AEAD encryption. Both the decryption techniques are decrypting using their respective associated keys. A server must have a storage to store information pertaining to these algorithms and their corresponding keys such as private key, Key K, etc., to establish secure TLS communication with a client.

	Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT. https://datatracker.ietf.org/doc/html/rfc8446#

	The "extension_data" field of these extensions contains a SignatureSchemeList value: `enum {` `    /* RSASSA-PKCS1-v1_5 algorithms */` `    rsa_pkcs1_sha256(0x0401),` `    rsa_pkcs1_sha384(0x0501),` `    rsa_pkcs1_sha512(0x0601),` `    /* ECDSA algorithms */` `    ecdsa_secp256r1_sha256(0x0403),` `    ecdsa_secp384r1_sha384(0x0503),` `    ecdsa_secp521r1_sha512(0x0603),` `    /* RSASSA-PSS algorithms with public key OID rsaEncryption */` `    rsa_pss_rsae_sha256(0x0804),` `    rsa_pss_rsae_sha384(0x0805),` `    rsa_pss_rsae_sha512(0x0806),` `    /* EdDSA algorithms */` `    ed25519(0x0807),` `    ed448(0x0808),` `    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */` `    rsa_pss_pss_sha256(0x0809),` `    rsa_pss_pss_sha384(0x080a),` `    rsa_pss_pss_sha512(0x080b),` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	There is also a decryption function D that takes a ciphertext and a decryption key K_D, and reproduces the plaintext message. $D(C, K_D) = P$ In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster. In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B's encryption key, encrypt a message with it, and send the result to person B. Only someone with B's decryption key, namely only B, can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q, compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de = 1 \pmod{\varphi(n)}$. we know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$. Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is $\boxed{c = m^e \bmod n.}$ First encryption To decrypt, we compute $c^d \bmod n$ to obtain $c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$ The result of Exercise 3.13 tells us that $\boxed{m \equiv m^{1+k\varphi(n)} \pmod{n},}$ First decryption https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions `MD5`, `SHA-1`, and `SHA-256`. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	This specification defines the following cipher suites for use with TLS 1.3. ``` +------------------------------+-------------+ | Description | Value | +------------------------------+-------------+ | TLS_AES_128_GCM_SHA256 | {0x13,0x01} | | | | | TLS_AES_256_GCM_SHA384 | {0x13,0x02} | | | | | TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} | | | | | TLS_AES_128_CCM_SHA256 | {0x13,0x04} | | | | | TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} | +------------------------------+-------------+ ``` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	2.2. Authenticated Decryption The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure). https://datatracker.ietf.org/doc/html/rfc5116 The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1
29. The system of claim 21, wherein each encryption algorithm is a symmetric key system or an asymmetric key system.	The standard practices the method such that each encryption algorithm (e.g., signature encryption algorithm i.e., SHA256RSA, etc., and AEAD encryption algorithm i.e., TLS_AES_256_GCM_SHA384, etc.) is a symmetric key system (e.g., AEAD encryption algorithm, etc.) or an asymmetric key system (e.g., signature encryption algorithm). As shown below, the server comprises a memory storage to store messages for

	establishing secure TLS communication. the standard discloses multiple signature encryption algorithms for a first encryption and multiple AEAD encryption algorithms for the second encryption. A signature decryption algorithm utilizes a private key for decrypting the first bitstream encrypted with the signature encryption and an AEAD decryption algorithm uses a key K for decrypting the second bitstream encrypted with the AEAD encryption. The standard defines the signature encryption algorithm as an asymmetric cryptography algorithm and the AEAD encryption algorithm as the symmetric cryptography algorithm. Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT. https://datatracker.ietf.org/doc/html/rfc8446# Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated. Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], the Elliptic Curve Digital Signature Algorithm (ECDSA) [ECDSA], or the Edwards-Curve Digital Signature Algorithm (EdDSA) [RFC8032]) or a symmetric pre-shared key (PSK). https://datatracker.ietf.org/doc/html/rfc8446#section-4

	cipher_suites: A list of the symmetric cipher options supported by the client, specifically the record protection algorithm (including secret key length) and a hash to be used with HKDF, in descending order of client preference. Values are defined in Appendix B.4. If the list contains cipher suites that the server does not recognize, support, or wish to use, the server MUST ignore those cipher suites and process the remaining ones as usual. If the client is attempting a PSK key establishment, it SHOULD advertise at least one cipher suite indicating a Hash associated with the PSK. https://datatracker.ietf.org/doc/html/rfc8446#section-4

	The "extension_data" field of these extensions contains a SignatureSchemeList value: `enum {` `    /* RSASSA-PKCS1-v1_5 algorithms */` `    rsa_pkcs1_sha256(0x0401),` `    rsa_pkcs1_sha384(0x0501),` `    rsa_pkcs1_sha512(0x0601),` `    /* ECDSA algorithms */` `    ecdsa_secp256r1_sha256(0x0403),` `    ecdsa_secp384r1_sha384(0x0503),` `    ecdsa_secp521r1_sha512(0x0603),` `    /* RSASSA-PSS algorithms with public key OID rsaEncryption */` `    rsa_pss_rsae_sha256(0x0804),` `    rsa_pss_rsae_sha384(0x0805),` `    rsa_pss_rsae_sha512(0x0806),` `    /* EdDSA algorithms */` `    ed25519(0x0807),` `    ed448(0x0808),` `    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */` `    rsa_pss_pss_sha256(0x0809),` `    rsa_pss_pss_sha384(0x080a),` `    rsa_pss_pss_sha512(0x080b),` https://datatracker.ietf.org/doc/html/rfc8446#section-1

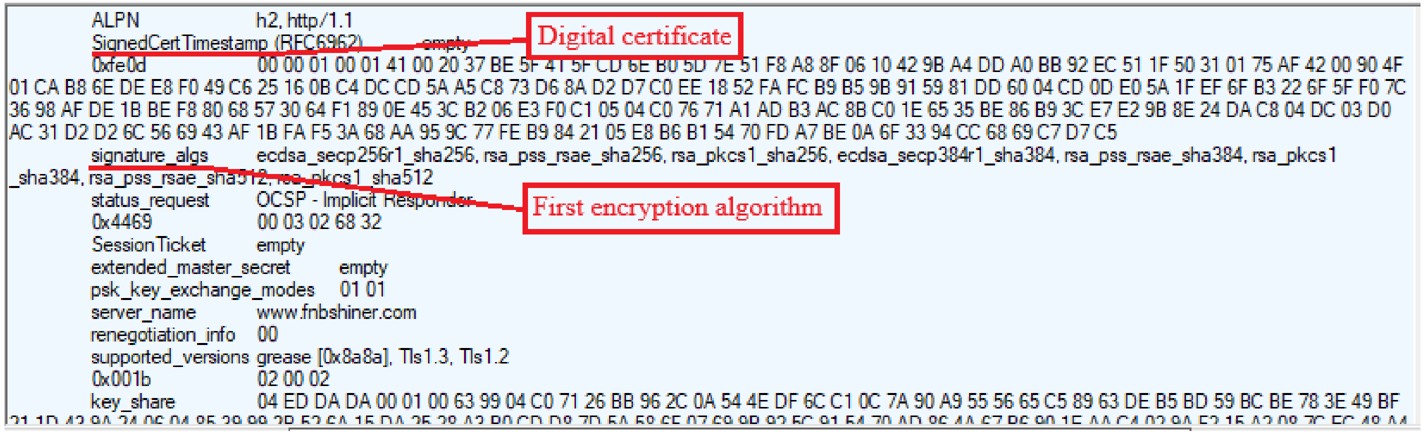
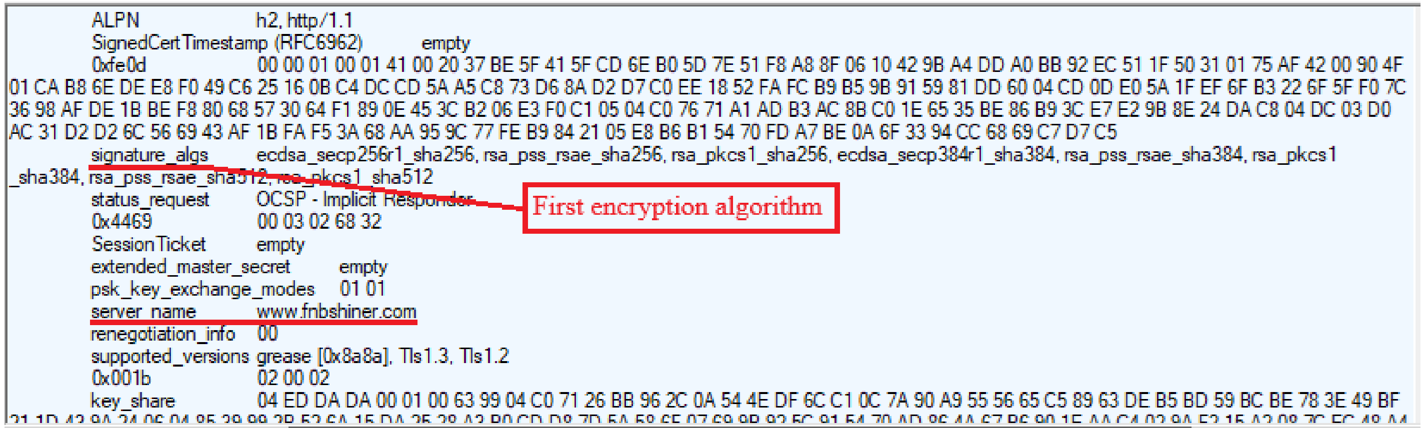
	There is also a decryption function D that takes a ciphertext and a decryption key K_D, and reproduces the plaintext message. $D(C, K_D) = P$ In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster. In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B's encryption key, encrypt a message with it, and send the result to person B. Only someone with B's decryption key, namely only B, can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

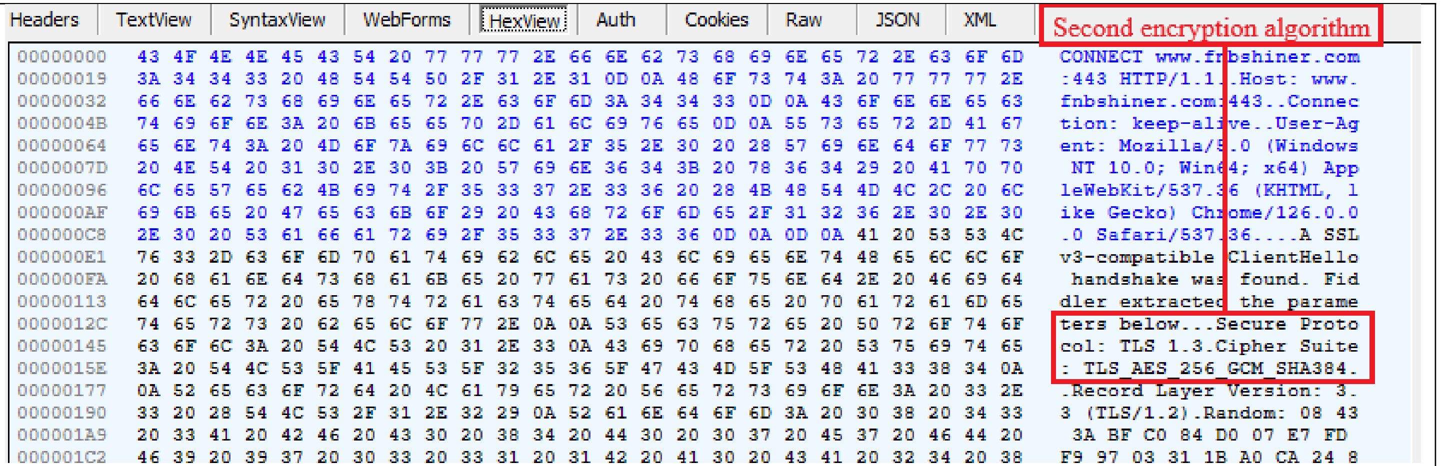
	The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	This specification defines the following cipher suites for use with TLS 1.3. | Description | Value | | TLS_AES_128_GCM_SHA256 | {0x13,0x01} | | TLS_AES_256_GCM_SHA384 | {0x13,0x02} | | TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} | | TLS_AES_128_CCM_SHA256 | {0x13,0x04} | | TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} | https://datatracker.ietf.org/doc/html/rfc8446#section-1

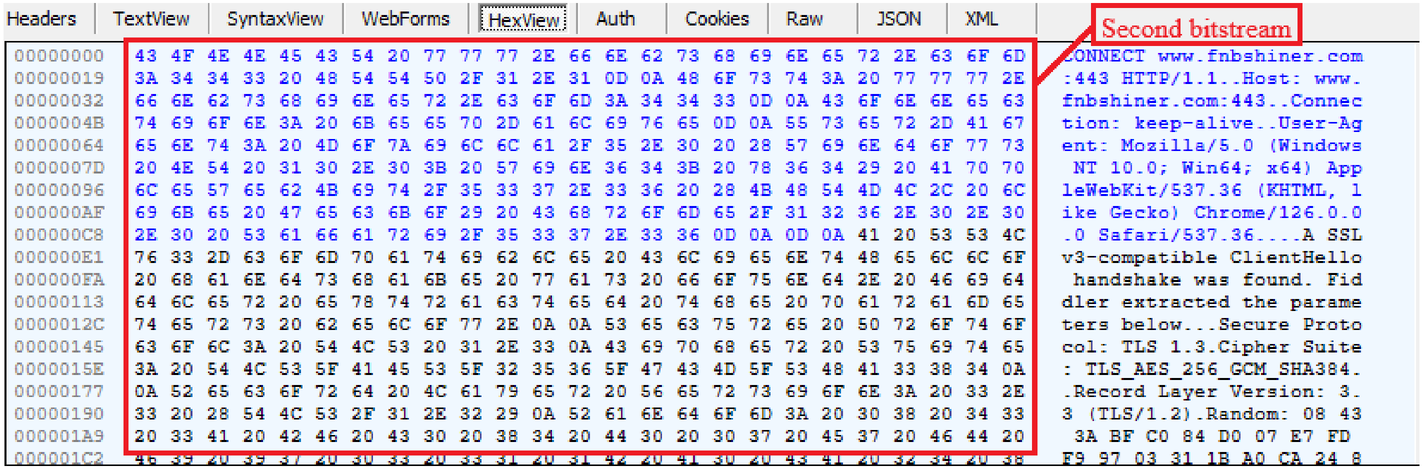
	The authenticated encryption operation has four inputs, each of which is an octet string: A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom. A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in Section 3.2, and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces. A plaintext P, which contains the data to be encrypted and authenticated. The associated data A, which contains the data to be authenticated, but not encrypted. https://datatracker.ietf.org/doc/html/rfc5116

	2.2. Authenticated Decryption The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure). https://datatracker.ietf.org/doc/html/rfc5116 The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1
30. The system of claim 21, further operable for associating a first Message Authentication Code (MAC) or first digital signature with each	The standard practices associating a first Message Authentication Code (MAC) (e.g., message authentication code with hashing function) or first digital signature with each encrypted bit stream (e.g., encrypted bit stream with the signature encryption algorithm i.e., SHA256RSA, etc., and encrypted bitstream with the AEAD encryption algorithm i.e., TLS_AES_256_GCM_SHA384, etc.). As shown below, the standard discloses a hashing function with each of the encryption

encrypted bit stream.	algorithm. It performs a message authentication code with the utilized hashing function.  *Source: Fiddler Capture*  *Source: Fiddler Capture*

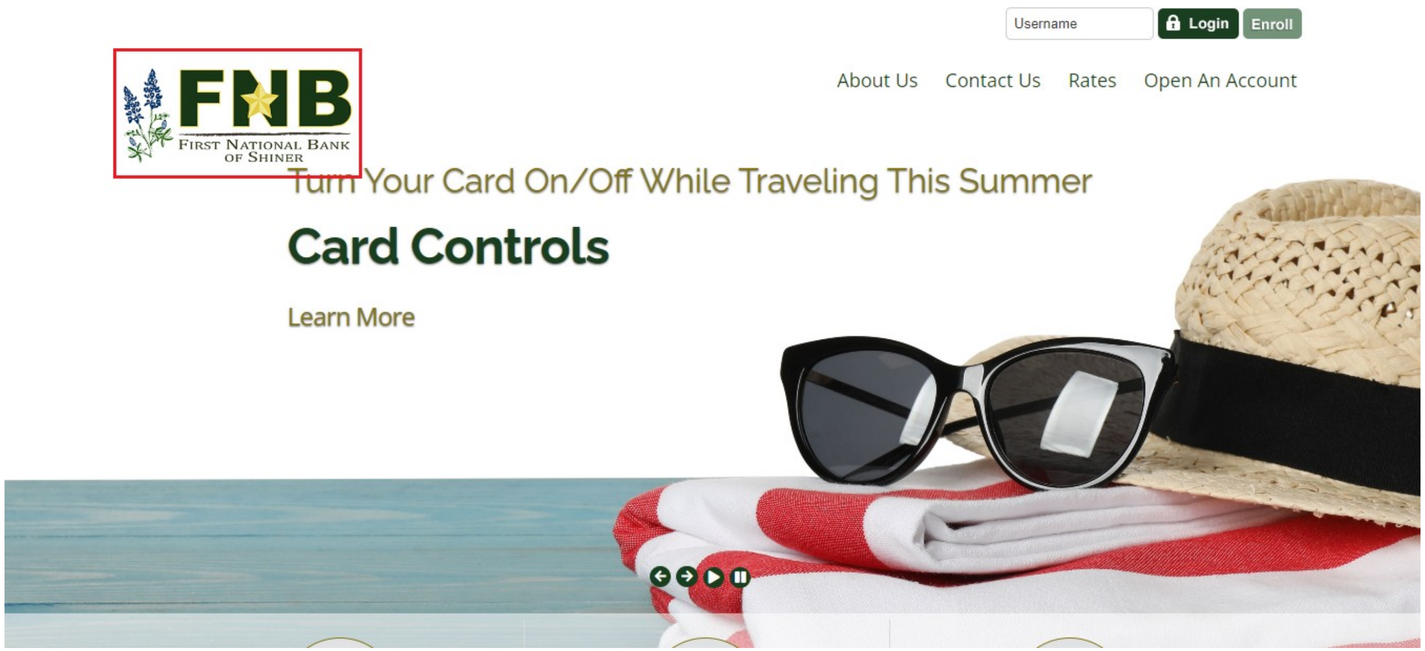


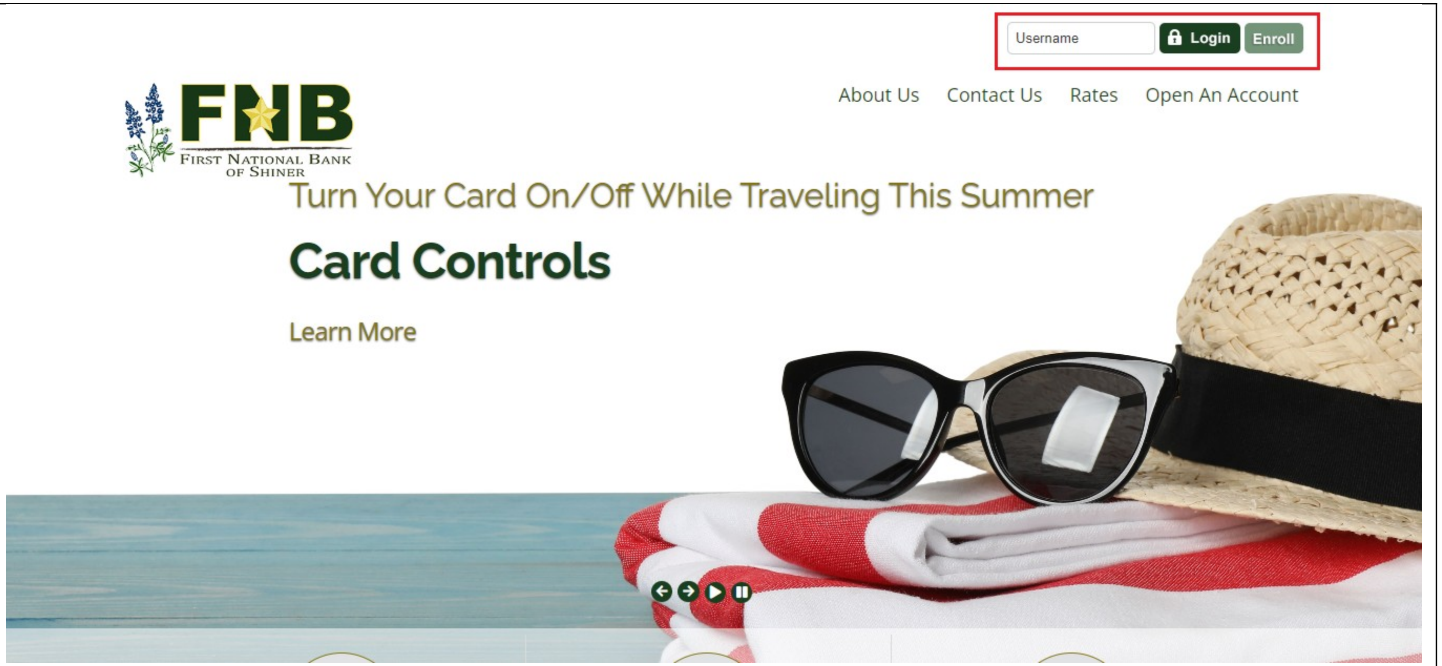
Source: Fiddler Capture

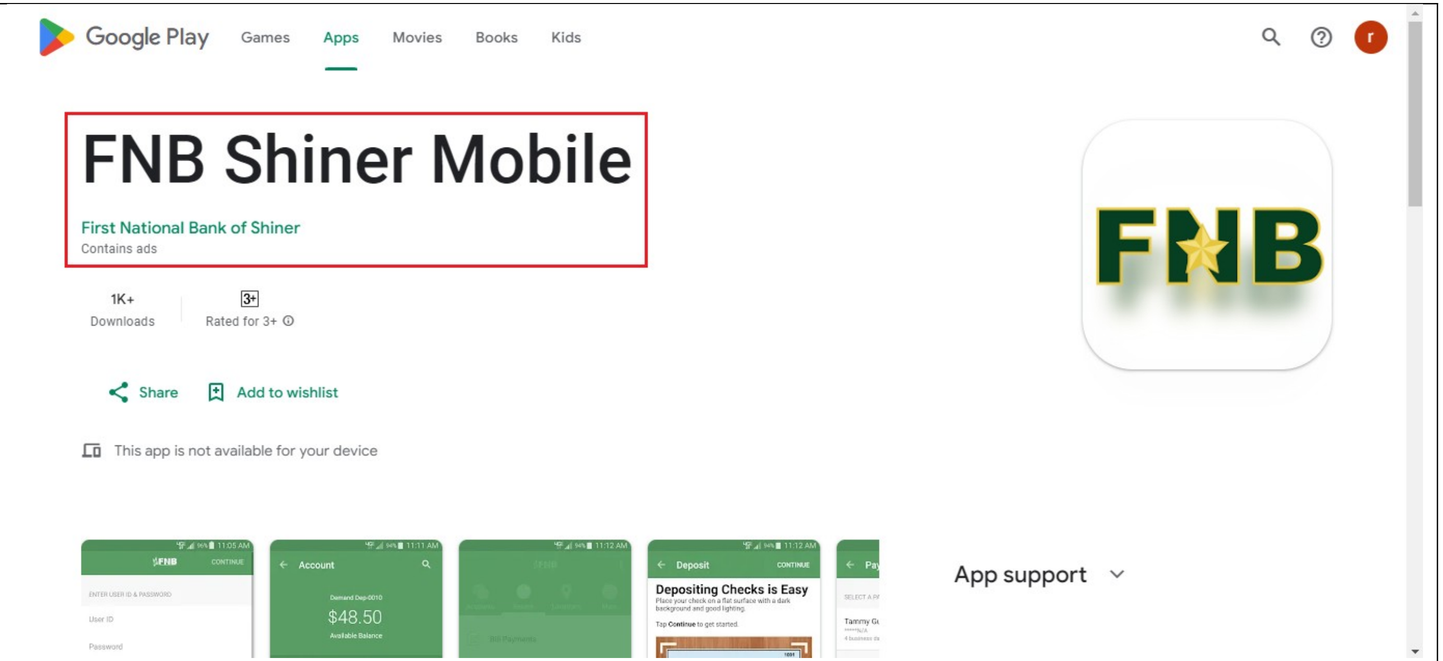


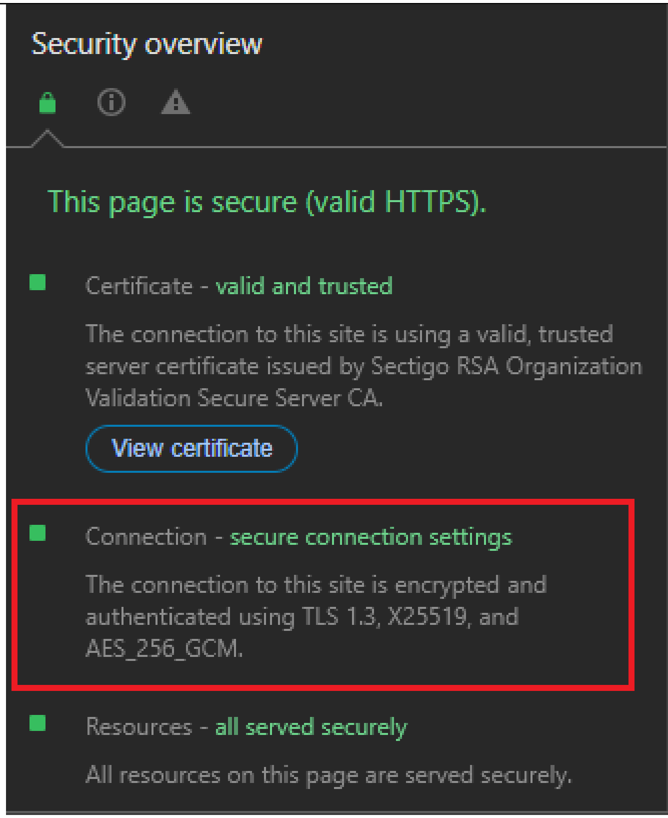
Source: Fiddler Capture

	The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf The list of supported symmetric encryption algorithms has been pruned of all algorithms that are considered legacy. Those that remain are all Authenticated Encryption with Associated Data (AEAD) algorithms. The cipher suite concept has been changed to separate the authentication and key exchange mechanisms from the record protection algorithm (including secret key length) and a hash to be used with both the key derivation function and handshake message authentication code (MAC). https://datatracker.ietf.org/doc/html/rfc8446#section-4
37. A computer storage device for a recursive	The accused instrumentality utilizes a computer storage device (e.g., a memory of the server of the accused instrumentality) for a recursive security protocol (e.g., TLS 1.3

security protocol for protecting digital content, comprising instructions executable by a processor for performing the steps of:	security protocol) for protecting digital content (e.g., digital certificate related to the accused instrumentality), comprising instructions executable by a processor (e.g., a processor of the server of the accused instrumentality). The accused instrumentality utilizes TLS 1.3 security protocol (hereinafter "the standard") for communicating content such as digital certificate, application data, etc., with a client. The standard provides a two-level encryption security. It encrypts a plaintext with a first encryption technique and generates a ciphertext. Further, it encrypts the ciphertext with a second encryption technique i.e., recursive encryption security.  https://www.fnbshiner.com/

	 https://www.fnbshiner.com/

	 https://play.google.com/store/apps/details?id=com.mfoundry.mb.android.mb_113106833&hl=en_US



https://www.fnbshiner.com/

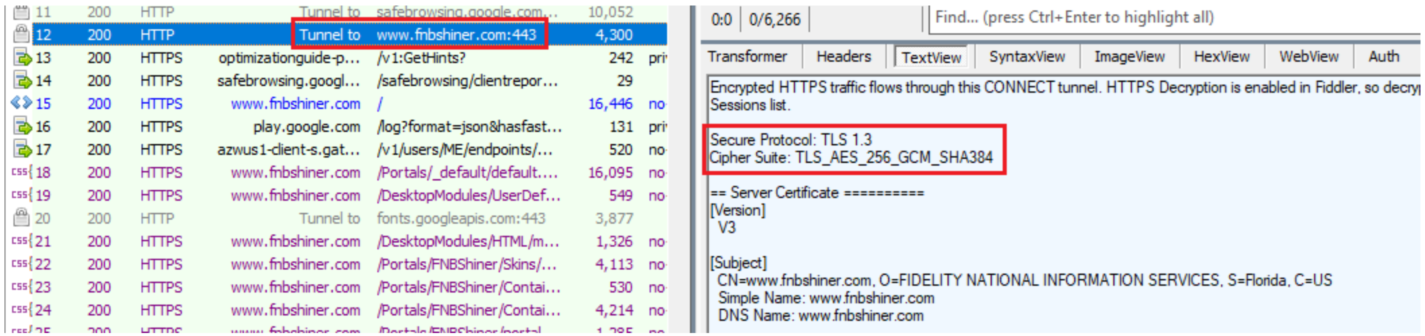
The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

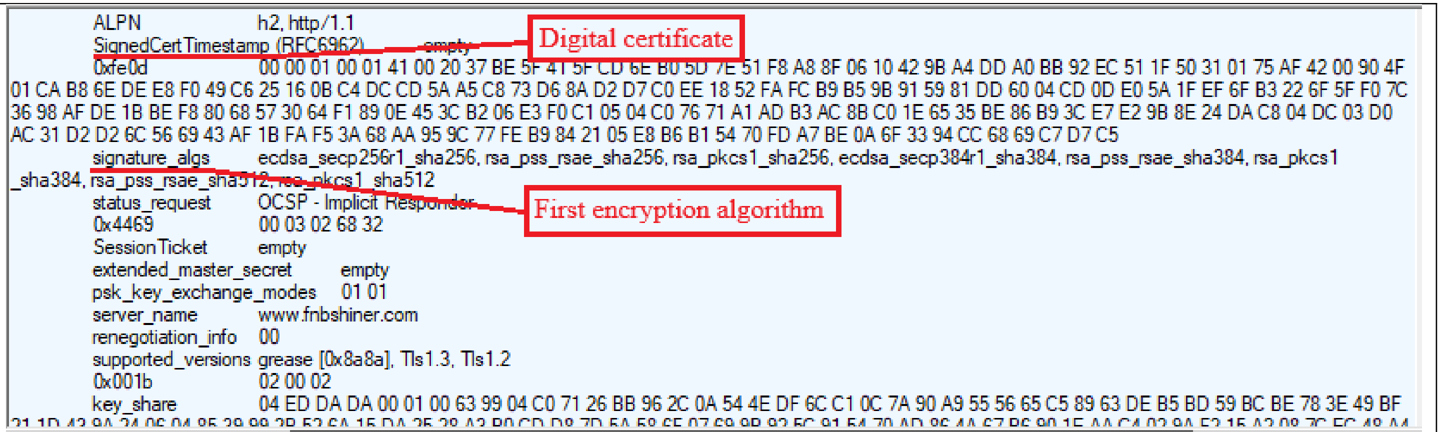
This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

https://datatracker.ietf.org/doc/html/rfc8446

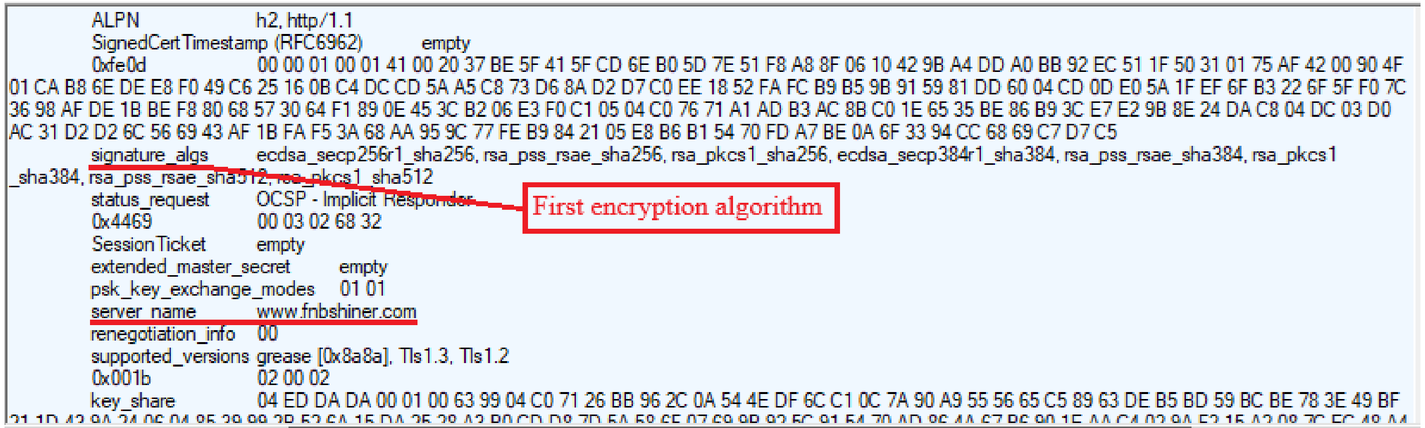
As shown below, the accused instrumentality utilizes a two-level algorithm security. It utilizes the SHA256RSA encryption algorithm as a first encryption algorithm i.e., signature encryption algorithm and the TLS_AES_256_GCM_SHA384 encryption algorithm as a second encryption algorithm i.e., AEAD encryption algorithm.



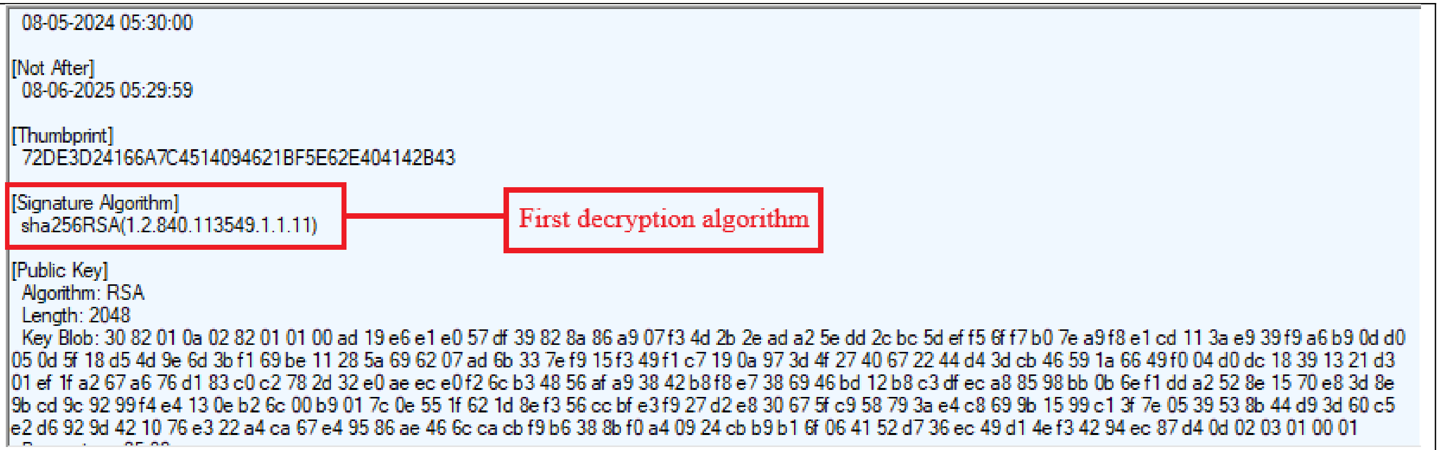
Source: Fiddler Capture



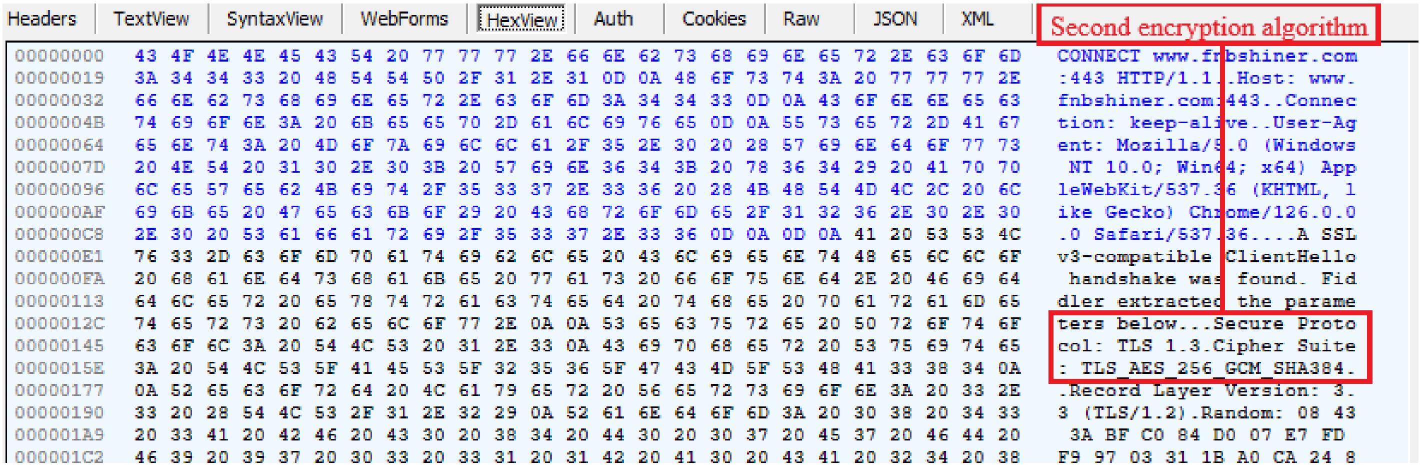
Source: Fiddler Capture



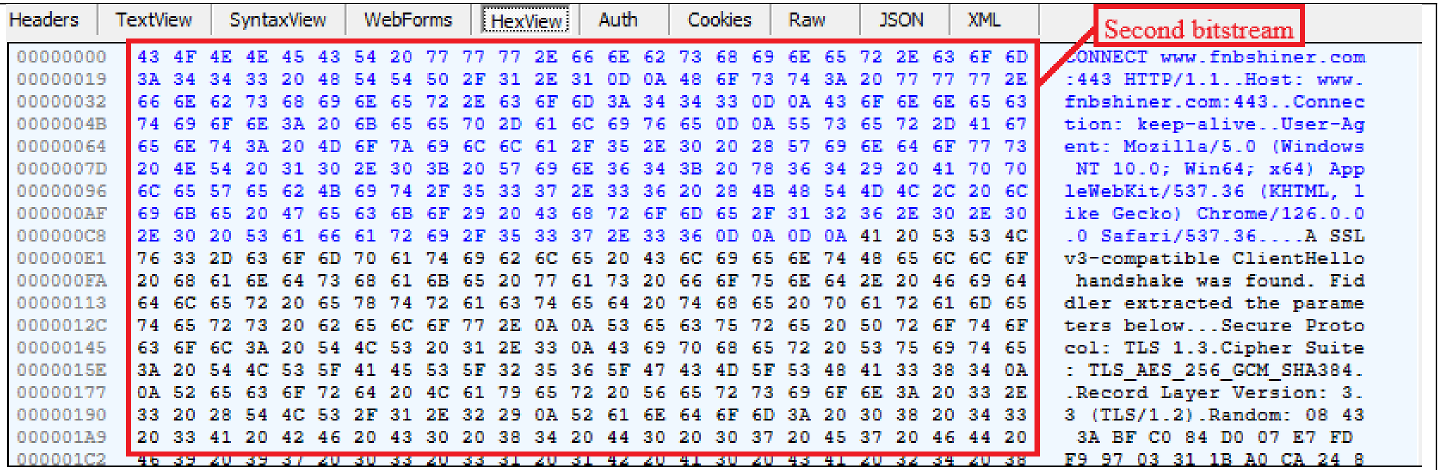
Source: Fiddler Capture



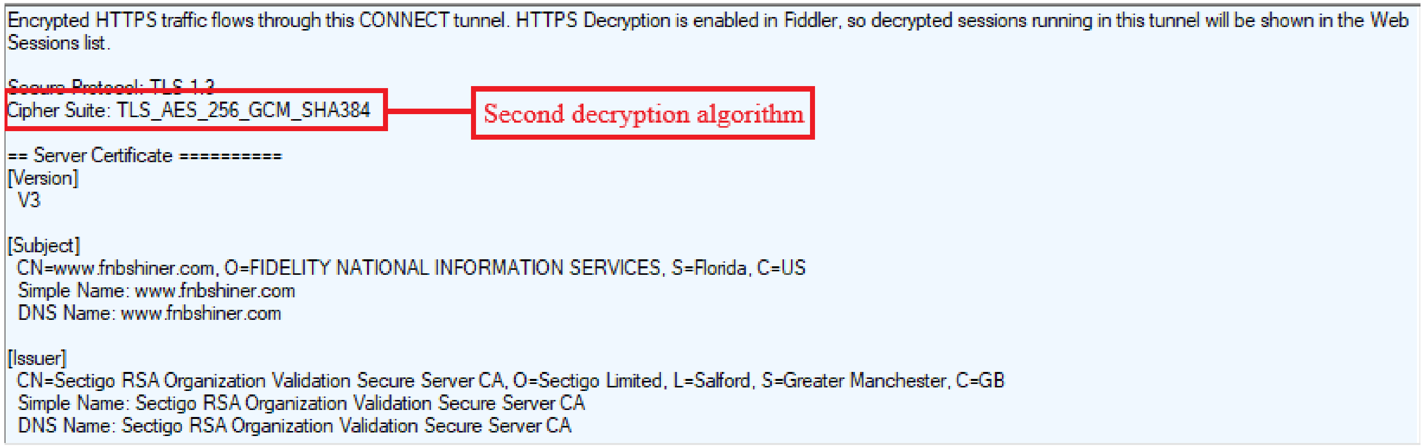
Source: Fiddler Capture



Source: Fiddler Capture

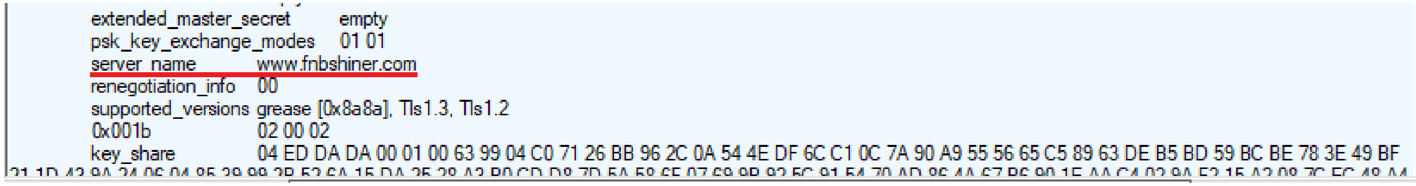


Source: Fiddler Capture



Source: Fiddler Capture

As shown below, the server of the accused instrumentality comprises a processor to execute instructions and a memory storage to store instructions for performing the operations defined by the standard.

	 extended_master_secret empty psk_key_exchange_modes 01 01 server_name www.fnbshiner.com renegotiation_info 00 supported_versions grease [0x8a8a], Tls1.3, Tls1.2 0x001b 02 00 02 key_share 04 ED DA DA 00 01 00 63 99 04 C0 71 26 BB 96 2C 0A 54 4E DF 6C C1 0C 7A 90 A9 55 56 65 C5 89 63 DE B5 BD 59 BC BE 78 3E 49 BF *Source: Fiddler Capture*

	Tech Accelerator **Server hardware guide: Architecture, products and management** **2. Processor** The CPU -- or simply processor -- is a complex micro-circuitry device that serves as the foundation of all computer operations. It supports hundreds of possible commands hardwired into hundreds of millions of transistors to process low-level software instructions -- microcode -- and data and derive a desired logical or mathematical result. The processor works closely with memory, which both holds the software instructions and data to be processed as well as the results or output of those processor operations. https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms

	Tech Accelerator **Server hardware guide: Architecture, products and management** **3. Random access memory** RAM is the main type of memory in a computing system. RAM holds the software instructions and data needed by the processor, along with any output from the processor, such as data to be moved to a storage device. Thus, RAM works very closely with the processor and must match the processor's incredible speed and performance. This kind of fast memory is usually termed dynamic RAM, and several DRAM variations are available for servers. https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms

	Tech Accelerator **Server hardware guide: Architecture, products and management** **4. Hard disk drive** This hardware is responsible for reading, writing and positioning of the hard disk, which is one technology for data storage on server hardware. Developed at IBM in 1953, the hard disk drive (HDD) has evolved over time from the size of a refrigerator to the standard 2.5-inch and 3.5-inch form factors. https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT.

	https://datatracker.ietf.org/doc/html/rfc8446# The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm. As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

	5. Record Protocol The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients. TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see Appendix D.4). https://datatracker.ietf.org/doc/html/rfc8446#section-1 **2. Protocol Overview** Negotiating encryption algorithm The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic. https://datatracker.ietf.org/doc/html/rfc8446

	TLS consists of two primary components: - A handshake protocol (Section 4) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack. Negotiating encryption algos - A record protocol (Section 5) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys. https://datatracker.ietf.org/doc/html/rfc8446 All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4. Authentication Messages As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [sender]_handshake_traffic_secret. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	Figure 1 below shows the basic full TLS handshake:

```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

Digital Content

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.1.1. Cryptographic Negotiation In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello: - A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports. Second encryption - A "supported_groups" (Section 4.2.7) extension which indicates the (EC)DHE groups which the client supports and a "key_share" (Section 4.2.8) extension which contains (EC)DHE shares for some or all of these groups. - A "signature_algorithms" (Section 4.2.3) extension which indicates the signature algorithms which the client can accept. A "signature_algorithms_cert" extension (Section 4.2.3) may also be added to indicate certificate-specific signature algorithms. First encryption - A "pre_shared_key" (Section 4.2.11) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" (Section 4.2.9) extension which indicates the key exchange modes that may be used with PSKs. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2. Certificate This message conveys the endpoint's certificate chain to the peer. The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK). The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2.3. Client Certificate Selection The following rules apply to certificates sent by the client: - The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]). - If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs. - The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS. - If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.3. Certificate Verify This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message. Structure of this message: `struct {` `   SignatureScheme algorithm;` `   opaque signature<0..2^16-1>;` `} CertificateVerify;` The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely: https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {
    opaque certificate_request_context<0..2^8-1>;
    Extension extensions<2..2^16-1>;
} CertificateRequest;
```

https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages. extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions. https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message. In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.2.3. Signature Algorithms TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2). The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	The "extension_data" field of these extensions contains a SignatureSchemeList value: `enum {` `/* RSASSA-PKCS1-v1_5 algorithms */` `rsa_pkcs1_sha256(0x0401),` `rsa_pkcs1_sha384(0x0501),` `rsa_pkcs1_sha512(0x0601),` `/* ECDSA algorithms */` `ecdsa_secp256r1_sha256(0x0403),` `ecdsa_secp384r1_sha384(0x0503),` `ecdsa_secp521r1_sha512(0x0603),` `/* RSASSA-PSS algorithms with public key OID rsaEncryption */` `rsa_pss_rsae_sha256(0x0804),` `rsa_pss_rsae_sha384(0x0805),` `rsa_pss_rsae_sha512(0x0806),` `/* EdDSA algorithms */` `ed25519(0x0807),` `ed448(0x0808),` `/* RSASSA-PSS algorithms with public key OID RSASSA-PSS */` `rsa_pss_pss_sha256(0x0809),` `rsa_pss_pss_sha384(0x080a),` `rsa_pss_pss_sha512(0x080b),` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	Introduction The primary goal of TLS is to provide a secure channel between two communicating peers; the only requirement from the underlying transport is a reliable, in-order data stream. Specifically, the secure channel should provide the following properties: First encryption - Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated. Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], the Elliptic Curve Digital Signature Algorithm (ECDSA) [ECDSA], or the Edwards-Curve Digital Signature Algorithm (EdDSA) [RFC8032]) or a symmetric pre-shared key (PSK). - Confidentiality: Data sent over the channel after establishment is only visible to the endpoints. TLS does not hide the length of the data it transmits, though endpoints are able to pad TLS records in order to obscure lengths and improve protection against traffic analysis techniques. - Integrity: Data sent over the channel after establishment cannot be modified by attackers without detection. https://datatracker.ietf.org/doc/html/rfc8446

	5.1. Record Layer The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^14 bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2. Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that: - Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **5.2. Record Payload Protection** The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding. https://datatracker.ietf.org/doc/html/rfc8446#section-1

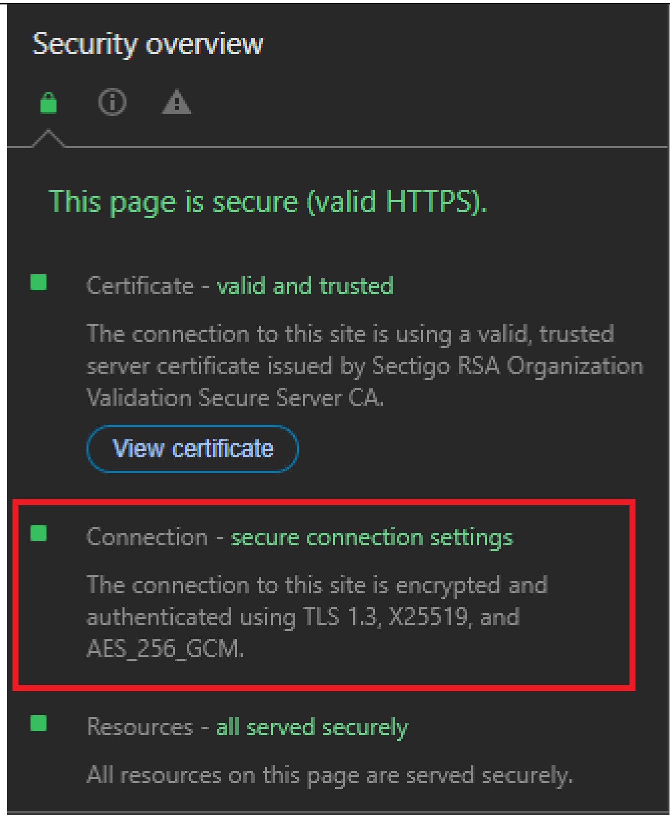
	AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see Section 5.3), and the additional data input is the record header. I.e., `additional_data = TLSCiphertext.opaque_type ||` `TLSCiphertext.legacy_record_version ||` `TLSCiphertext.length` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case. https://datatracker.ietf.org/doc/html/rfc5116

	2.1. Authenticated Encryption The authenticated encryption operation has four inputs, each of which is an octet string: A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom. A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in Section 3.2, and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces. A plaintext P, which contains the data to be encrypted and authenticated. The associated data A, which contains the data to be authenticated, but not encrypted. https://datatracker.ietf.org/doc/html/rfc5116

	The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1 Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [RFC5116], Section 4). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows: https://datatracker.ietf.org/doc/html/rfc8446#section-1

	This specification defines the following cipher suites for use with TLS 1.3. ``` +------------------------------+-------------+ | Description | Value | +------------------------------+-------------+ | TLS_AES_128_GCM_SHA256 | {0x13,0x01} | | | | | TLS_AES_256_GCM_SHA384 | {0x13,0x02} | | | | | TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} | | | | | TLS_AES_128_CCM_SHA256 | {0x13,0x04} | | | | | TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} | +------------------------------+-------------+ ``` https://datatracker.ietf.org/doc/html/rfc8446#section-1
encrypting a bit stream with a first encryption algorithm;	The standard practices encrypting a bitstream (e.g., bitstream of digital certificate) with a first encryption algorithm (e.g., signature encryption algorithm i.e., SHA256RSA encryption algorithm). The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA encryption algorithm) and generates a ciphertext.



https://www.fnbshiner.com/

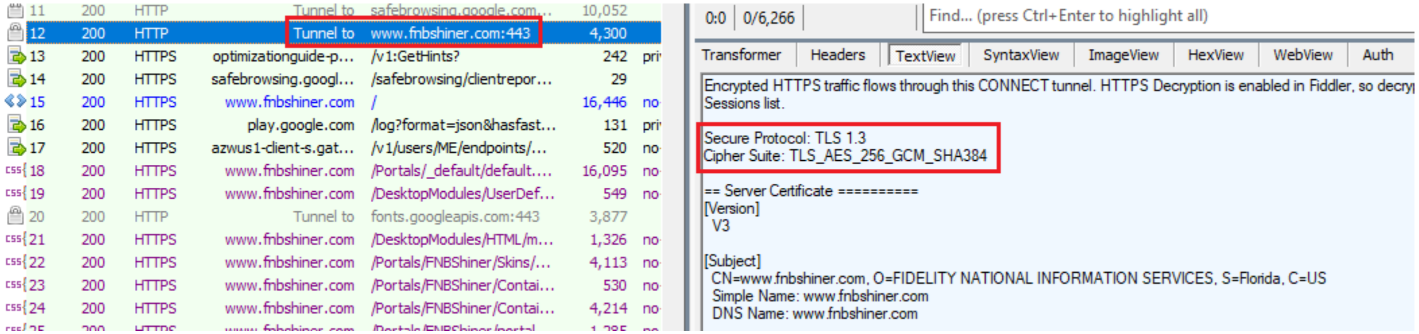
The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

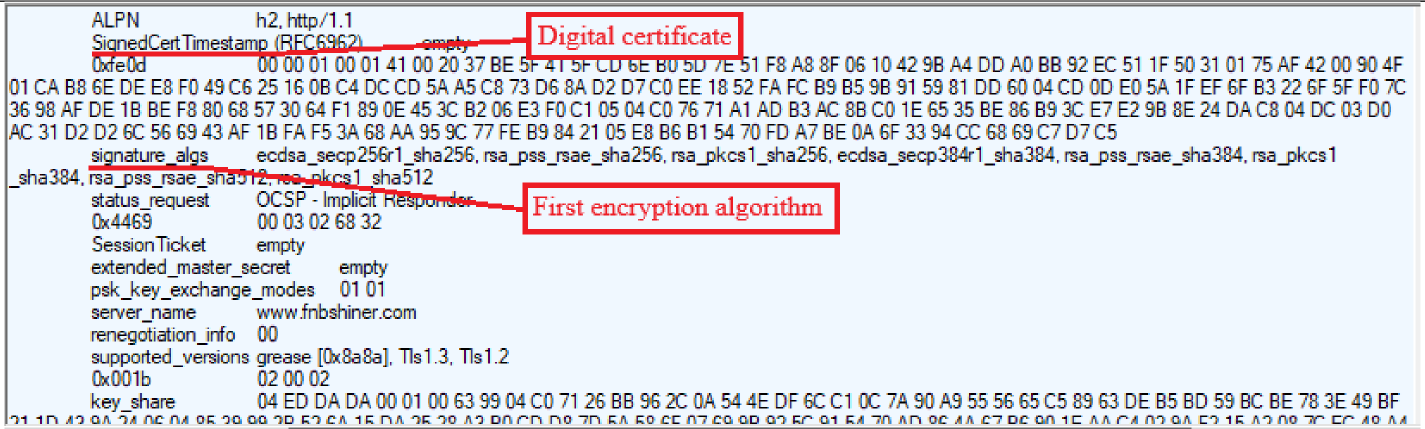
This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

https://datatracker.ietf.org/doc/html/rfc8446

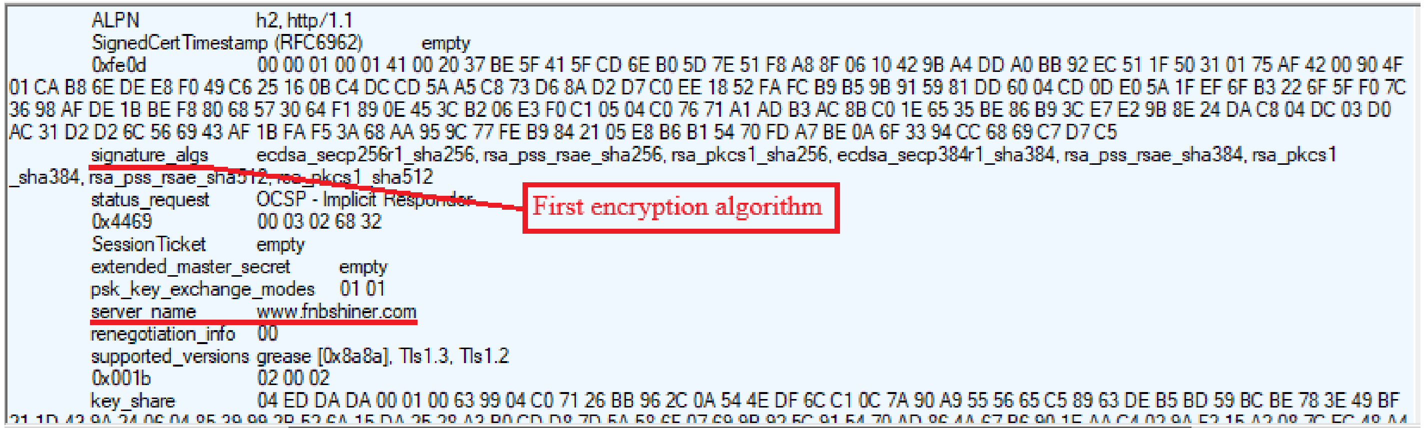
As shown below, the accused instrumentality discloses the signature encryption algorithm.



Source: Fiddler Capture



Source: Fiddler Capture



Source: Fiddler Capture

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are

	encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm. As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server. **5. Record Protocol** The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients. TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see Appendix D.4). https://datatracker.ietf.org/doc/html/rfc8446#section-1

	2. Protocol Overview Negotiating encryption algorithm The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic. https://datatracker.ietf.org/doc/html/rfc8446 TLS consists of two primary components: - A handshake protocol (Section 4) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack. Negotiating encryption algos - A record protocol (Section 5) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys. https://datatracker.ietf.org/doc/html/rfc8446

	All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **4.4. Authentication Messages** As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [sender]_handshake_traffic_secret. https://datatracker.ietf.org/doc/html/rfc8446#section-1

Figure 1 below shows the basic full TLS handshake:

```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

Digital Content

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.1.1. Cryptographic Negotiation In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello: - A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports. - A "supported_groups" (Section 4.2.7) extension which indicates the (EC)DHE groups which the client supports and a "key_share" (Section 4.2.8) extension which contains (EC)DHE shares for some or all of these groups. First encryption - A "signature_algorithms" (Section 4.2.3) extension which indicates the signature algorithms which the client can accept. A "signature_algorithms_cert" extension (Section 4.2.3) may also be added to indicate certificate-specific signature algorithms. - A "pre_shared_key" (Section 4.2.11) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" (Section 4.2.9) extension which indicates the key exchange modes that may be used with PSKs. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2. Certificate This message conveys the endpoint's certificate chain to the peer. The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK). The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2.3. Client Certificate Selection The following rules apply to certificates sent by the client: - The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]). - If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs. - The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS. - If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.3. Certificate Verify This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message. Structure of this message: `struct {` `SignatureScheme algorithm;` `opaque signature<0..2^16-1>;` `} CertificateVerify;` The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely: https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	4.3.2. Certificate Request A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions. Structure of this message: `struct {` `opaque certificate_request_context<0..2^8-1>;` `Extension extensions<2..2^16-1>;` `} CertificateRequest;` https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages. extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions. https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message. In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.2.3. Signature Algorithms TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2). The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	The "extension_data" field of these extensions contains a SignatureSchemeList value: `enum {` `    /* RSASSA-PKCS1-v1_5 algorithms */` `    rsa_pkcs1_sha256(0x0401),` `    rsa_pkcs1_sha384(0x0501),` `    rsa_pkcs1_sha512(0x0601),` `    /* ECDSA algorithms */` `    ecdsa_secp256r1_sha256(0x0403),` `    ecdsa_secp384r1_sha384(0x0503),` `    ecdsa_secp521r1_sha512(0x0603),` `    /* RSASSA-PSS algorithms with public key OID rsaEncryption */` `    rsa_pss_rsae_sha256(0x0804),` `    rsa_pss_rsae_sha384(0x0805),` `    rsa_pss_rsae_sha512(0x0806),` `    /* EdDSA algorithms */` `    ed25519(0x0807),` `    ed448(0x0808),` `    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */` `    rsa_pss_pss_sha256(0x0809),` `    rsa_pss_pss_sha384(0x080a),` `    rsa_pss_pss_sha512(0x080b),` https://datatracker.ietf.org/doc/html/rfc8446#section-1

<table>
<tr><td></td><td>

Introduction

The primary goal of TLS is to provide a secure channel between two communicating peers; the only requirement from the underlying transport is a reliable, in-order data stream. Specifically, the secure channel should provide the following properties:

First encryption

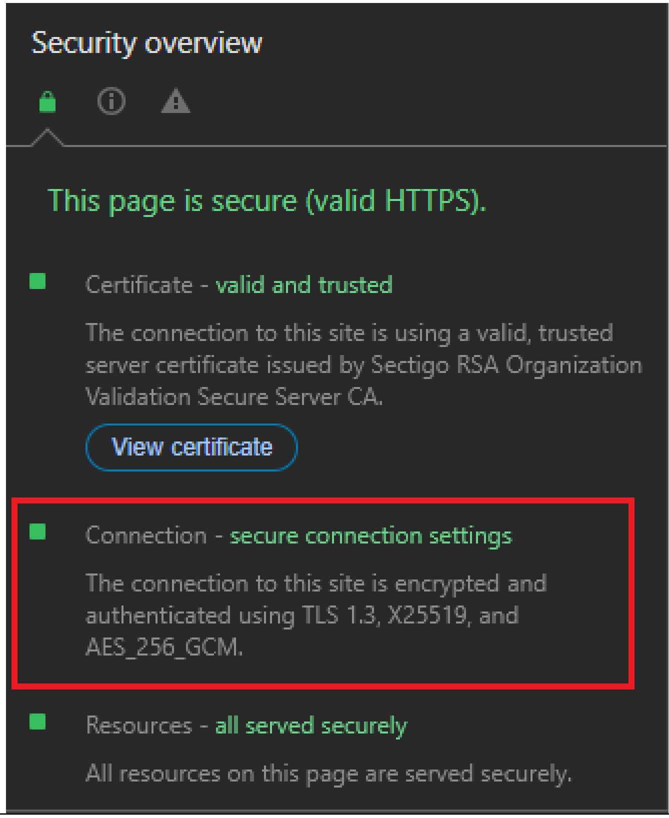
- Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated. Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], the Elliptic Curve Digital Signature Algorithm (ECDSA) [ECDSA], or the Edwards-Curve Digital Signature Algorithm (EdDSA) [RFC8032]) or a symmetric pre-shared key (PSK).

- Confidentiality: Data sent over the channel after establishment is only visible to the endpoints. TLS does not hide the length of the data it transmits, though endpoints are able to pad TLS records in order to obscure lengths and improve protection against traffic analysis techniques.

- Integrity: Data sent over the channel after establishment cannot be modified by attackers without detection.

https://datatracker.ietf.org/doc/html/rfc8446
</td></tr>
<tr><td>associating a first decryption algorithm with the encrypted bit stream;</td><td>The standard practices associating a first decryption algorithm (e.g., signature decryption algorithm i.e., SHA256RSA decryption algorithm) with the encrypted bit stream (e.g., encrypted certificate with signature encryption algorithm).

The standard practices providing a two-level encryption security for data</td></tr>
</table>

	communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA encryption algorithm) and generates a ciphertext. The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it. The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate. 

	https://www.fnbshiner.com/

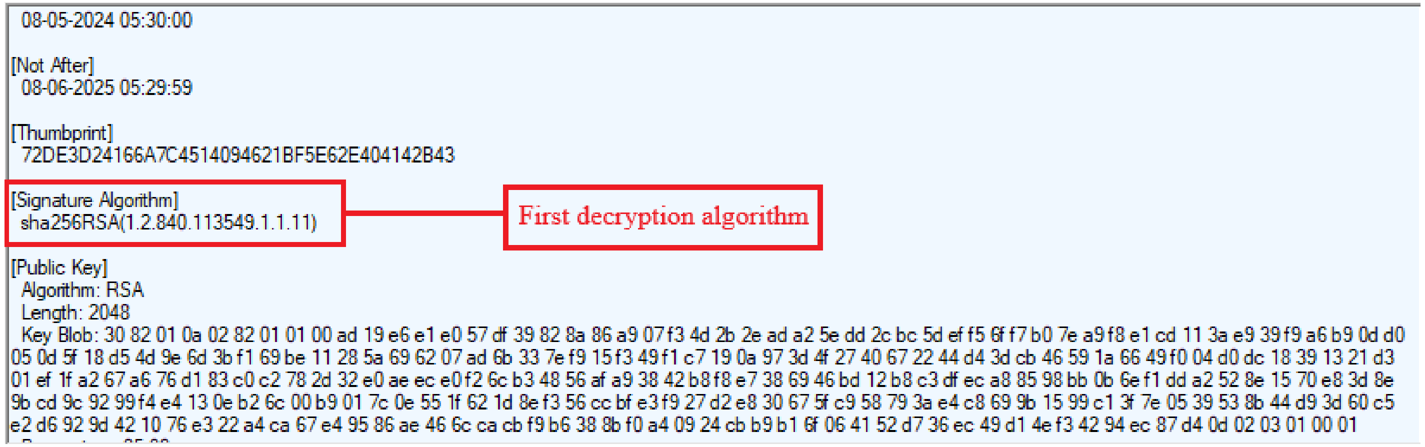
The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

https://datatracker.ietf.org/doc/html/rfc8446

As shown below, the accused instrumentality discloses the signature decryption algorithm.


08-05-2024 05:30:00

[Not After]
08-06-2025 05:29:59

[Thumbprint]
72DE3D24166A7C4514094621BF5E62E404142B43

[Signature Algorithm]
sha256RSA(1.2.840.113549.1.1.11)

First decryption algorithm

[Public Key]
Algorithm: RSA
Length: 2048
Key Blob: 30 82 01 0a 02 82 01 01 00 ad 19 e6 e1 e0 57 df 39 82 8a 86 a9 07 f3 4d 2b 2e ad a2 5e dd 2c bc 5d ef f5 6f f7 b0 7e a9 f8 e1 cd 11 3a e9 39 f9 a6 b9 0d d0 05 0d 5f 18 d5 4d 9e 6d 3b f1 69 be 11 28 5a 69 62 07 ad 6b 33 7e f9 15 f3 49 f1 c7 19 0a 97 3d 4f 27 40 67 22 44 d4 3d cb 46 59 1a 66 49 f0 04 d0 dc 18 39 13 21 d3 01 ef 1f a2 67 a6 76 d1 83 c0 c2 78 2d 32 e0 ae ec e0 f2 6c b3 48 56 af a9 38 42 b8 f8 e7 38 69 46 bd 12 b8 c3 df ec a8 85 98 bb 0b 6e f1 dd a2 52 8e 15 70 e8 3d 8e 9b cd 9c 92 99 f4 e4 13 0e b2 6c 00 b9 01 7c 0e 55 1f 62 1d 8e f3 56 cc bf e3 f9 27 d2 e8 30 67 5f c9 58 79 3a e4 c8 69 9b 15 99 c1 3f 7e 05 39 53 8b 44 d9 3d 60 c5 e2 d6 92 9d 42 10 76 e3 22 a4 ca 67 e4 95 86 ae 46 6c ca cb f9 b6 38 8b f0 a4 09 24 cb b9 b1 6f 06 41 52 d7 36 ec 49 d1 4e f3 42 94 ec 87 d4 0d 02 03 01 00 01

Source: Fiddler Capture

	OID description First decryption algorithm identifier **OID**: {iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs-1(1) sha256WithRSAEncryption(11)} (ASN.1 notation) 1.2.840.113549.1.1.11 (dot notation) /ISO/Member-Body/US/113549/1/1/11 (OID-IRI notation) **Description**: Public-Key Cryptography Standards (PKCS) #1 version 1.5 signature algorithm with Secure Hash Algorithm 256 (SHA256) and Rivest, Shamir and Adleman (RSA) encryption http://oid-info.com/get/1.2.840.113549.1.1.11 `-- When the following OIDs are used in an AlgorithmIdentifier, the` `-- parameters MUST be present and MUST be NULL.` `sha224WithRSAEncryption  OBJECT IDENTIFIER  ::=  { pkcs-1 14 }` `sha256WithRSAEncryption  OBJECT IDENTIFIER  ::=  { pkcs-1 11 }` `sha384WithRSAEncryption  OBJECT IDENTIFIER  ::=  { pkcs-1 12 }` `sha512WithRSAEncryption  OBJECT IDENTIFIER  ::=  { pkcs-1 13 }` https://www.ietf.org/rfc/rfc4055.txt

```
   Figure 1 below shows the basic full TLS handshake:

       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

Digital Content

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.3. Certificate Verify This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message. Structure of this message: `struct {` `SignatureScheme algorithm;` `opaque signature<0..2^16-1>;` `} CertificateVerify;` The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely: https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages. extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions. https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	- RSASSA-PSS signature schemes are defined in Section 4.2.3. - The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello. - The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates. https://datatracker.ietf.org/doc/html/rfc8446#section-1 The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm. As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

	5. Record Protocol The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients. TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see Appendix D.4). https://datatracker.ietf.org/doc/html/rfc8446#section-1 **2. Protocol Overview** Negotiating encryption algorithm The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic. https://datatracker.ietf.org/doc/html/rfc8446

	TLS consists of two primary components: - A handshake protocol (Section 4) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack. Negotiating encryption algos - A record protocol (Section 5) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys. https://datatracker.ietf.org/doc/html/rfc8446 All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4. Authentication Messages As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [sender]_handshake_traffic_secret. https://datatracker.ietf.org/doc/html/rfc8446#section-1

```
   Figure 1 below shows the basic full TLS handshake:

       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

Digital Content

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.1.1. Cryptographic Negotiation In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello: - A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports. - A "supported_groups" (Section 4.2.7) extension which indicates the (EC)DHE groups which the client supports and a "key_share" (Section 4.2.8) extension which contains (EC)DHE shares for some or all of these groups. First encryption - A "signature_algorithms" (Section 4.2.3) extension which indicates the signature algorithms which the client can accept. A "signature_algorithms_cert" extension (Section 4.2.3) may also be added to indicate certificate-specific signature algorithms. - A "pre_shared_key" (Section 4.2.11) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" (Section 4.2.9) extension which indicates the key exchange modes that may be used with PSKs. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2. Certificate This message conveys the endpoint's certificate chain to the peer. The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK). The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2.3. Client Certificate Selection The following rules apply to certificates sent by the client: - The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]). - If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs. - The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS. - If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5. https://datatracker.ietf.org/doc/html/rfc8446#section-1

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {
    SignatureScheme algorithm;
    opaque signature<0..2^16-1>;
} CertificateVerify;
```

First decryption algorithm information

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	4.3.2. Certificate Request A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions. Structure of this message: `struct {` `    opaque certificate_request_context<0..2^8-1>;` `    Extension extensions<2..2^16-1>;` `} CertificateRequest;` https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages. extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions. https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	- RSASSA-PSS signature schemes are defined in Section 4.2.3. - The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello. - The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates. https://datatracker.ietf.org/doc/html/rfc8446#section-1 If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message. In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.2.3. Signature Algorithms TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2). The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension. https://datatracker.ietf.org/doc/html/rfc8446#section-1

<table>
<tr>
<td></td>
<td>

```
The "extension_data" field of these extensions contains a
SignatureSchemeList value:

   enum {
       /* RSASSA-PKCS1-v1_5 algorithms */
       rsa_pkcs1_sha256(0x0401),
       rsa_pkcs1_sha384(0x0501),
       rsa_pkcs1_sha512(0x0601),

       /* ECDSA algorithms */
       ecdsa_secp256r1_sha256(0x0403),
       ecdsa_secp384r1_sha384(0x0503),
       ecdsa_secp521r1_sha512(0x0603),

       /* RSASSA-PSS algorithms with public key OID rsaEncryption */
       rsa_pss_rsae_sha256(0x0804),
       rsa_pss_rsae_sha384(0x0805),
       rsa_pss_rsae_sha512(0x0806),

       /* EdDSA algorithms */
       ed25519(0x0807),
       ed448(0x0808),

       /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */
       rsa_pss_pss_sha256(0x0809),
       rsa_pss_pss_sha384(0x080a),
       rsa_pss_pss_sha512(0x080b),
```

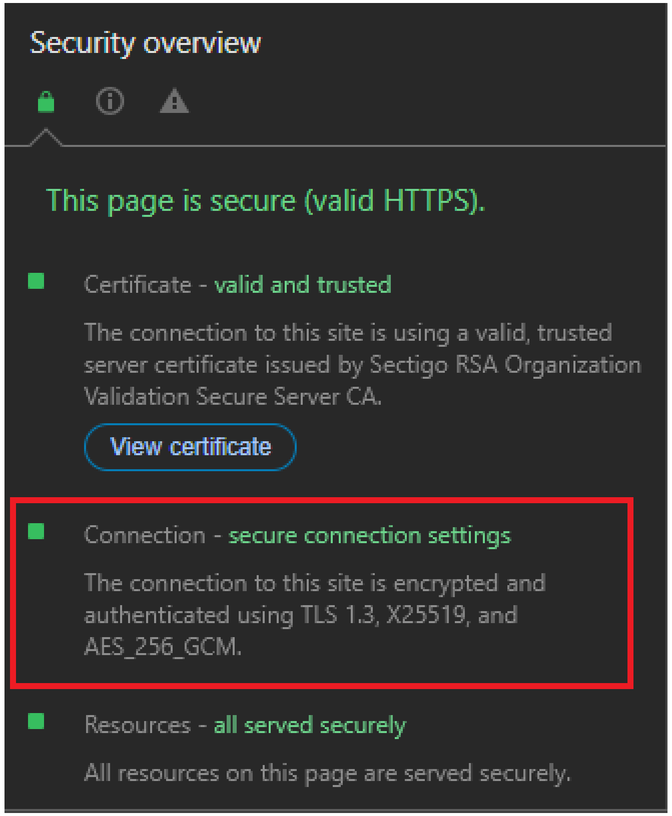
https://datatracker.ietf.org/doc/html/rfc8446#section-1

As shown below, the receiving party will be able to decrypt the encrypted message with the provided signature decryption algorithm information i.e., SHA-256 RSA decryption algorithm.

</td>
</tr>
</table>

	We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q, compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de = 1 \pmod{\varphi(n)}$. we know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$. Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is $\boxed{c = m^e \bmod n.}$ First encryption To decrypt, we compute $c^d \bmod n$ to obtain $$c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$$ The result of Exercise 3.13 tells us that $\boxed{m \equiv m^{1+k\varphi(n)} \pmod{n},}$ First decryption https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf
encrypting both the encrypted bit stream and the first decryption algorithm with a second encryption algorithm to yield a second bit stream;	The standard practices encrypting both the encrypted bit stream (e.g., encrypted digital certificate) and the first decryption algorithm (e.g., signature decryption algorithm) with a second encryption algorithm (e.g., cipher suit selected from one of the AEAD algorithms such as TLS_AES_256_GCM_SHA384, etc.) to yield a second bit stream (e.g., TLS ciphertext bitstream). The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA algorithm) and generates a ciphertext. The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it.

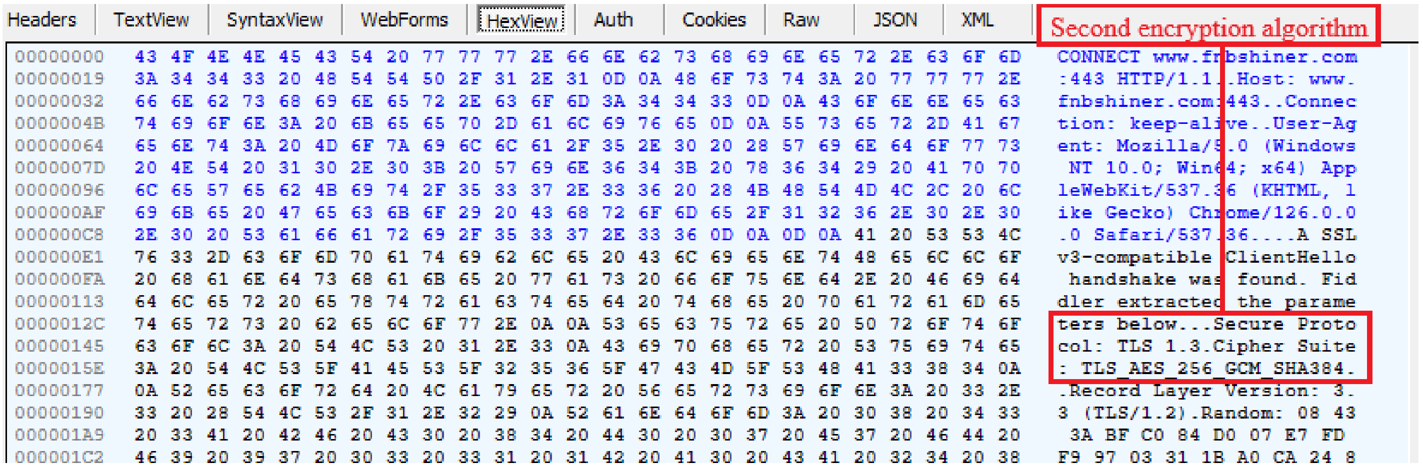
	The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate. The standard further practices encrypting the authentication message, including the encrypted digital certification and the certificate verify message, with a second decryption algorithm i.e., AEAD algorithm such as TLS_AES_256_GCM_SHA384, etc.  https://www.fnbshiner.com/

The Transport Layer Security (TLS) Protocol Version 1.3

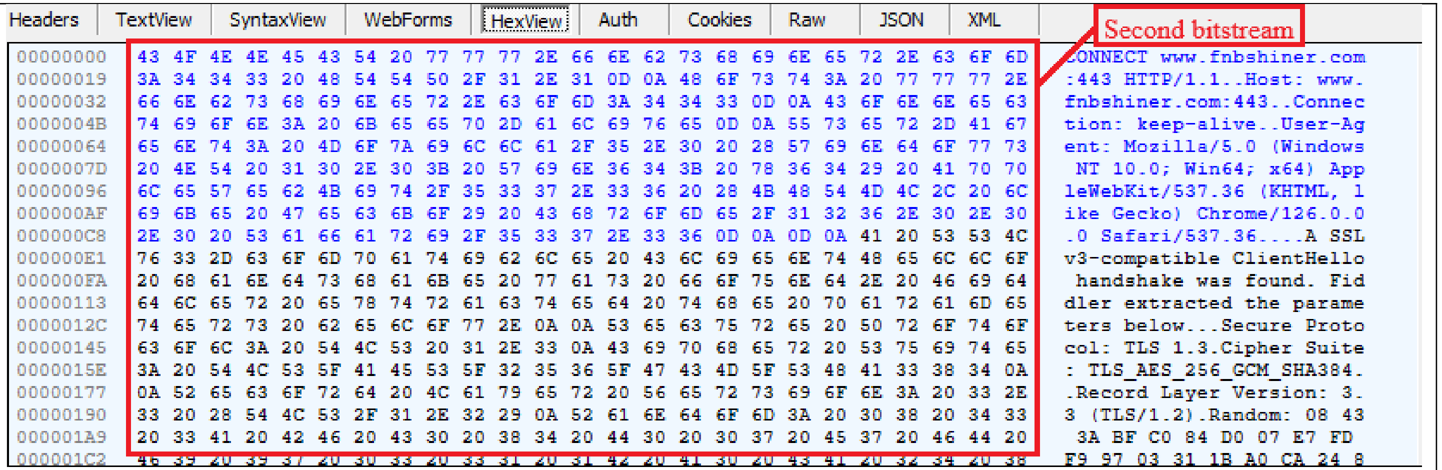
Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

https://datatracker.ietf.org/doc/html/rfc8446



Source: Fiddler Capture



Source: Fiddler Capture

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm.

As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message

	is communicated between the client and the server. **5. Record Protocol** The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients. TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see Appendix D.4). https://datatracker.ietf.org/doc/html/rfc8446#section-1 **2. Protocol Overview** Negotiating encryption algorithm The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic. https://datatracker.ietf.org/doc/html/rfc8446

	TLS consists of two primary components: - A handshake protocol (Section 4) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack. Negotiating encryption algos - A record protocol (Section 5) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys. https://datatracker.ietf.org/doc/html/rfc8446

	5.1. Record Layer The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^14 bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2. Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that: - Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **5.2. Record Payload Protection** The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see Section 5.3), and the additional data input is the record header. I.e., `additional_data = TLSCiphertext.opaque_type ||` `TLSCiphertext.legacy_record_version ||` `TLSCiphertext.length` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case. https://datatracker.ietf.org/doc/html/rfc5116

	2.1. Authenticated Encryption The authenticated encryption operation has four inputs, each of which is an octet string: A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom. A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in Section 3.2, and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces. A plaintext P, which contains the data to be encrypted and authenticated. The associated data A, which contains the data to be authenticated, but not encrypted. https://datatracker.ietf.org/doc/html/rfc5116

	The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1 Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [RFC5116], Section 4). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows: https://datatracker.ietf.org/doc/html/rfc8446#section-1

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

https://datatracker.ietf.org/doc/html/rfc8446#section-1

All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection.

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4. Authentication Messages As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [sender]_handshake_traffic_secret. https://datatracker.ietf.org/doc/html/rfc8446#section-1

Figure 1 below shows the basic full TLS handshake:

```
        Client                                           Server

 Key  ^ ClientHello
 Exch | + key_share*
      | + signature_algorithms*
      | + psk_key_exchange_modes*
      v + pre_shared_key*       -------->
                                                   ServerHello  ^ Key
                                                  + key_share*  | Exch
                                             + pre_shared_key*  v
                                         {EncryptedExtensions}  ^  Server
                                         {CertificateRequest*}  v  Params
                                                {Certificate*}  ^
                                          {CertificateVerify*}  | Auth
                                                    {Finished}  v
                                <--------  [Application Data*]
      ^ {Certificate*}
 Auth | {CertificateVerify*}
      v {Finished}              -------->
        [Application Data]      <------->  [Application Data]
```

Digital Content

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.1.1. Cryptographic Negotiation In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello: - A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports. Second encryption - A "supported_groups" (Section 4.2.7) extension which indicates the (EC)DHE groups which the client supports and a "key_share" (Section 4.2.8) extension which contains (EC)DHE shares for some or all of these groups. - A "signature_algorithms" (Section 4.2.3) extension which indicates the signature algorithms which the client can accept. A "signature_algorithms_cert" extension (Section 4.2.3) may also be added to indicate certificate-specific signature algorithms. First encryption - A "pre_shared_key" (Section 4.2.11) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" (Section 4.2.9) extension which indicates the key exchange modes that may be used with PSKs. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2. Certificate This message conveys the endpoint's certificate chain to the peer. The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK). The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2.3. Client Certificate Selection The following rules apply to certificates sent by the client: - The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]). - If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs. - The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS. - If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5. https://datatracker.ietf.org/doc/html/rfc8446#section-1

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {
    SignatureScheme algorithm;
    opaque signature<0..2^16-1>;
} CertificateVerify;
```

First decryption algorithm information

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	4.3.2. Certificate Request A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions. Structure of this message: `struct {` `opaque certificate_request_context<0..2^8-1>;` `Extension extensions<2..2^16-1>;` `} CertificateRequest;` https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages. extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions. https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	- RSASSA-PSS signature schemes are defined in Section 4.2.3. - The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello. - The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates. https://datatracker.ietf.org/doc/html/rfc8446#section-1 If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message. In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages. https://datatracker.ietf.org/doc/html/rfc8446#section-1

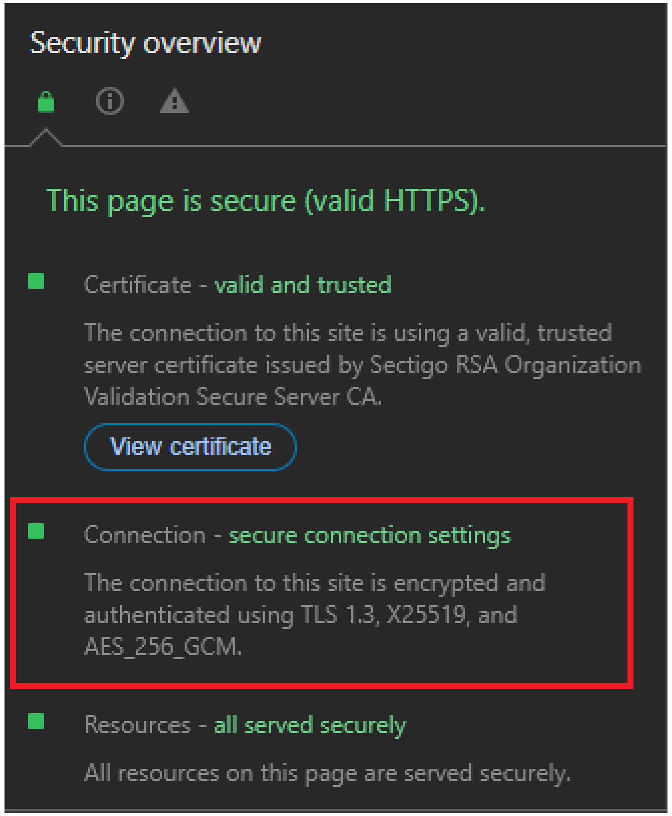
	4.2.3. Signature Algorithms TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2). The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension. https://datatracker.ietf.org/doc/html/rfc8446#section-1

<table>
<tr>
<td></td>
<td>

```
The "extension_data" field of these extensions contains a
SignatureSchemeList value:

   enum {
       /* RSASSA-PKCS1-v1_5 algorithms */
       rsa_pkcs1_sha256(0x0401),
       rsa_pkcs1_sha384(0x0501),
       rsa_pkcs1_sha512(0x0601),

       /* ECDSA algorithms */
       ecdsa_secp256r1_sha256(0x0403),
       ecdsa_secp384r1_sha384(0x0503),
       ecdsa_secp521r1_sha512(0x0603),

       /* RSASSA-PSS algorithms with public key OID rsaEncryption */
       rsa_pss_rsae_sha256(0x0804),
       rsa_pss_rsae_sha384(0x0805),
       rsa_pss_rsae_sha512(0x0806),

       /* EdDSA algorithms */
       ed25519(0x0807),
       ed448(0x0808),

       /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */
       rsa_pss_pss_sha256(0x0809),
       rsa_pss_pss_sha384(0x080a),
       rsa_pss_pss_sha512(0x080b),
```

https://datatracker.ietf.org/doc/html/rfc8446#section-1
</td>
</tr>
<tr>
<td>associating a second decryption algorithm with the second bit stream.</td>
<td>The standard practices associating a second decryption algorithm (e.g., cipher suit selected from one of the AEAD algorithms such as TLS_AES_256_GCM_SHA384, etc.) with the second bit stream (e.g., TLS ciphertext bitstream).

The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA algorithm) and generates a ciphertext.</td>
</tr>
</table>

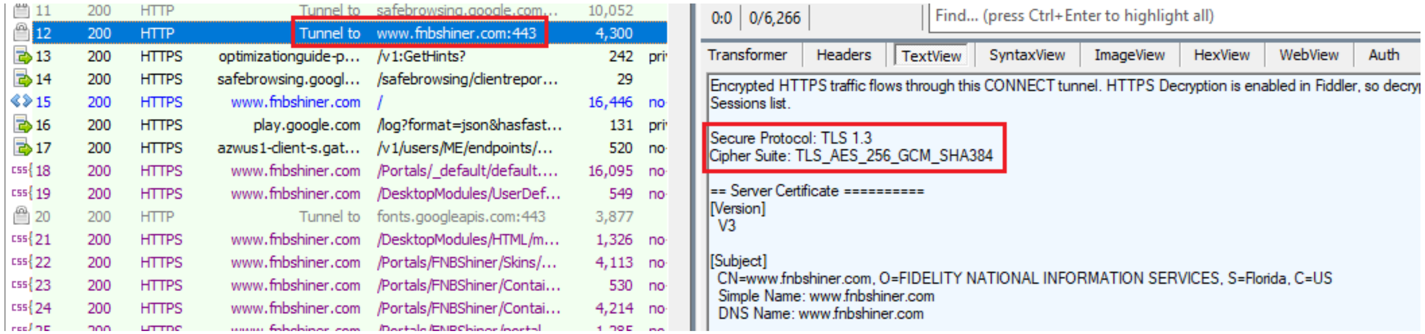
	The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it. The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate. The standard further practices encrypting the authentication message, including the encrypted digital certification and the certificate verify message, with a second decryption algorithm i.e., AEAD algorithm such as TLS_AES_256_GCM_SHA384, etc.  https://www.fnbshiner.com/

The Transport Layer Security (TLS) Protocol Version 1.3

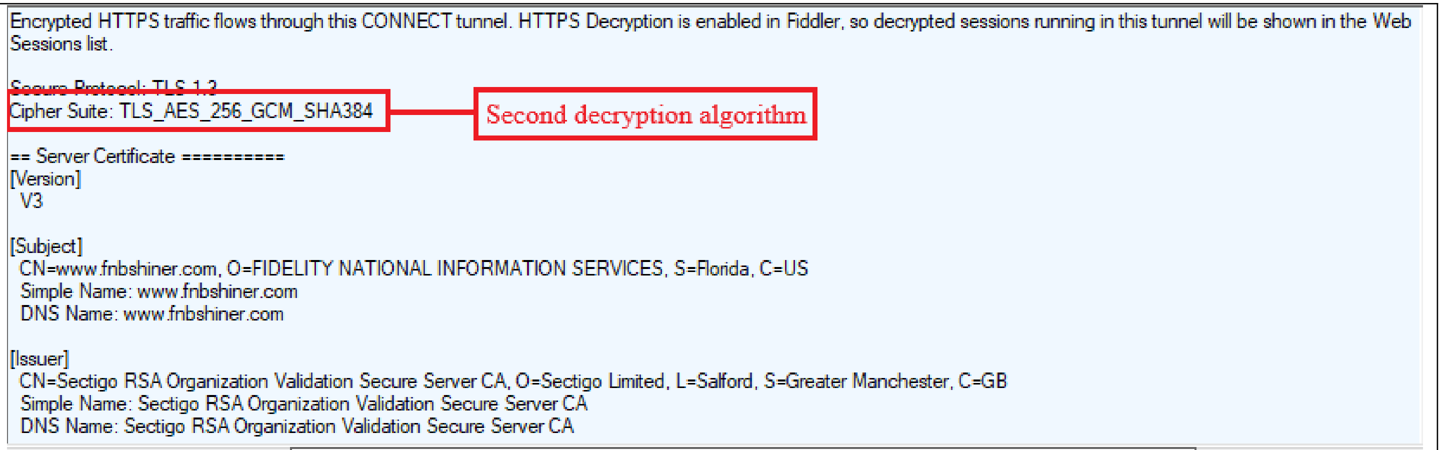
Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

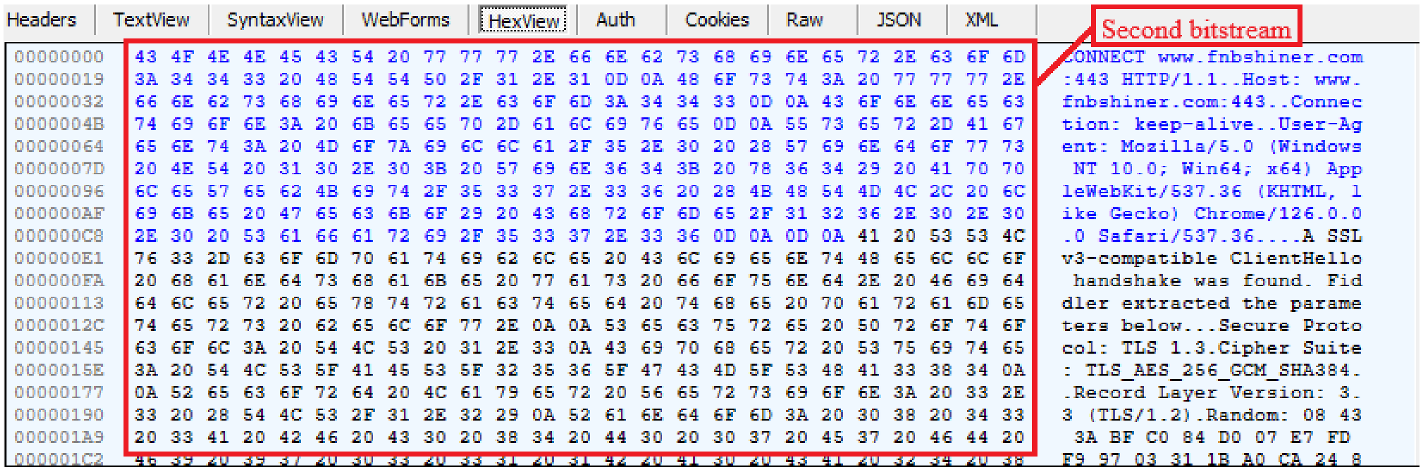
https://datatracker.ietf.org/doc/html/rfc8446



Source: Fiddler Capture



Source: Fiddler Capture



Source: Fiddler Capture

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for

	encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm. As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server. Further, the AEAD encrypted message comprises a ciphertext (e.g., encrypted ciphertext after the encryption by the second encryption algorithm), nonce (e.g., associating second decryption algo), key and associated data. The maximum length of nonce is a cipher suit specific element. The nonce and associated data are utilized in decryption of the AEAD encrypted message.

	5. Record Protocol The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients. TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see Appendix D.4). https://datatracker.ietf.org/doc/html/rfc8446#section-1 **2. Protocol Overview** Negotiating encryption algorithm The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic. https://datatracker.ietf.org/doc/html/rfc8446

	TLS consists of two primary components: - A handshake protocol (Section 4) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack. Negotiating encryption algos - A record protocol (Section 5) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys. https://datatracker.ietf.org/doc/html/rfc8446

	5.1. Record Layer The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^14 bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2. Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that: - Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **5.2. Record Payload Protection** The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see Section 5.3), and the additional data input is the record header. I.e., `additional_data = TLSCiphertext.opaque_type ||` `TLSCiphertext.legacy_record_version ||` `TLSCiphertext.length` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case. https://datatracker.ietf.org/doc/html/rfc5116

	2.1. Authenticated Encryption The authenticated encryption operation has four inputs, each of which is an octet string: A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom. A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in Section 3.2, and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces. A plaintext P, which contains the data to be encrypted and authenticated. The associated data A, which contains the data to be authenticated, but not encrypted. https://datatracker.ietf.org/doc/html/rfc5116

	2.2. Authenticated Decryption Second decryption algorithm The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure). https://datatracker.ietf.org/doc/html/rfc5116 The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [RFC5116], Section 4). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows: https://datatracker.ietf.org/doc/html/rfc8446#section-1 This specification defines the following cipher suites for use with TLS 1.3. | Description | Value | | TLS_AES_128_GCM_SHA256 | {0x13,0x01} | | TLS_AES_256_GCM_SHA384 | {0x13,0x02} | | TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} | | TLS_AES_128_CCM_SHA256 | {0x13,0x04} | | TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} | https://datatracker.ietf.org/doc/html/rfc8446#section-1

	All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **4.4. Authentication Messages** As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [sender]_handshake_traffic_secret. https://datatracker.ietf.org/doc/html/rfc8446#section-1

Figure 1 below shows the basic full TLS handshake:

```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

Digital Content

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2. Certificate This message conveys the endpoint's certificate chain to the peer. The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK). The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2.3. Client Certificate Selection The following rules apply to certificates sent by the client: - The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]). - If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs. - The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS. - If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.3. Certificate Verify This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message. Structure of this message: `struct {` `    SignatureScheme algorithm;` `    opaque signature<0..2^16-1>;` `} CertificateVerify;` The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely: https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	4.3.2. Certificate Request A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions. Structure of this message: `struct {` `opaque certificate_request_context<0..2^8-1>;` `Extension extensions<2..2^16-1>;` `} CertificateRequest;` https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages. extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions. https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	- RSASSA-PSS signature schemes are defined in Section 4.2.3. - The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello. - The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates. https://datatracker.ietf.org/doc/html/rfc8446#section-1 If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message. In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages. https://datatracker.ietf.org/doc/html/rfc8446#section-1

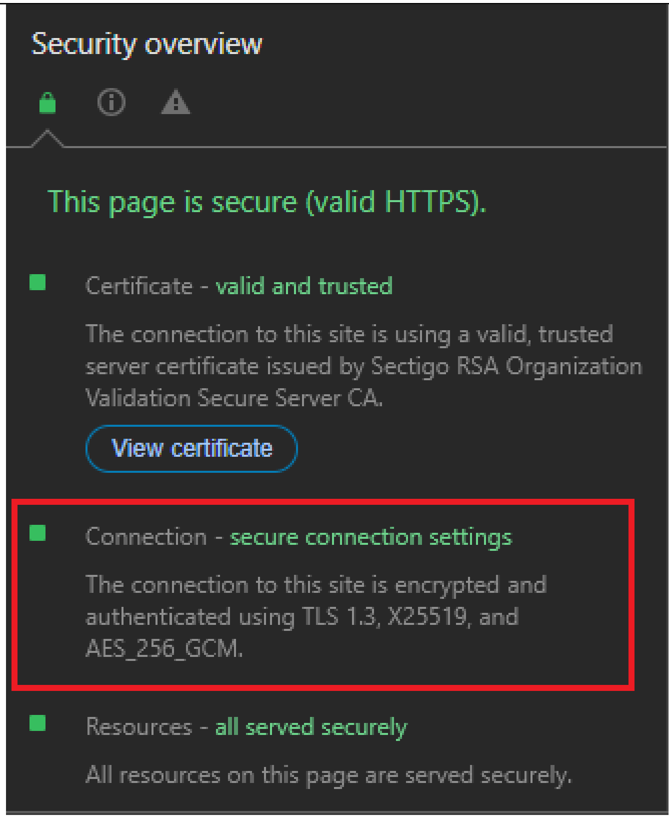
	4.2.3. Signature Algorithms TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2). The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension. https://datatracker.ietf.org/doc/html/rfc8446#section-1

<table>
<tr>
<td></td>
<td>

```
The "extension_data" field of these extensions contains a
SignatureSchemeList value:

   enum {
       /* RSASSA-PKCS1-v1_5 algorithms */
       rsa_pkcs1_sha256(0x0401),
       rsa_pkcs1_sha384(0x0501),
       rsa_pkcs1_sha512(0x0601),

       /* ECDSA algorithms */
       ecdsa_secp256r1_sha256(0x0403),
       ecdsa_secp384r1_sha384(0x0503),
       ecdsa_secp521r1_sha512(0x0603),

       /* RSASSA-PSS algorithms with public key OID rsaEncryption */
       rsa_pss_rsae_sha256(0x0804),
       rsa_pss_rsae_sha384(0x0805),
       rsa_pss_rsae_sha512(0x0806),

       /* EdDSA algorithms */
       ed25519(0x0807),
       ed448(0x0808),

       /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */
       rsa_pss_pss_sha256(0x0809),
       rsa_pss_pss_sha384(0x080a),
       rsa_pss_pss_sha512(0x080b),
```

https://datatracker.ietf.org/doc/html/rfc8446#section-1
</td>
</tr>
<tr>
<td>38. The software system or computer program of claim 37, further translatable for</td>
<td>The standard further discloses decrypting the first bit stream (e.g., encrypted digital certificate with signature encryption algorithm i.e., SHA-256 RSA, etc.) and the second bit stream (e.g., a second-level encryption with AEAD encryption algorithm such as TLS_AES_256_GCM_SHA384, etc.) with the first associated decryption</td>
</tr>
</table>

decrypting the first bit stream and the second bit stream with the first associated decryption algorithm and the second associated decryption algorithm wherein the decryption is accomplished by a target unit.	algorithm (e.g., signature decryption algorithm i.e., SHA-256 RSA, etc.) and the second associated decryption algorithm (e.g., cipher suit selected from one of the AEAD decryption algorithms such as TLS_AES_256_GCM_SHA384, etc.) wherein the decryption is accomplished by a target unit (e.g., a server of the accused instrumentality). The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA algorithm) and generates a ciphertext. The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it. The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate. The standard further practices encrypting the authentication message, including the encrypted digital certification and the certificate verify message, with a second decryption algorithm i.e., AEAD algorithm such as TLS_AES_256_GCM_SHA384, etc.

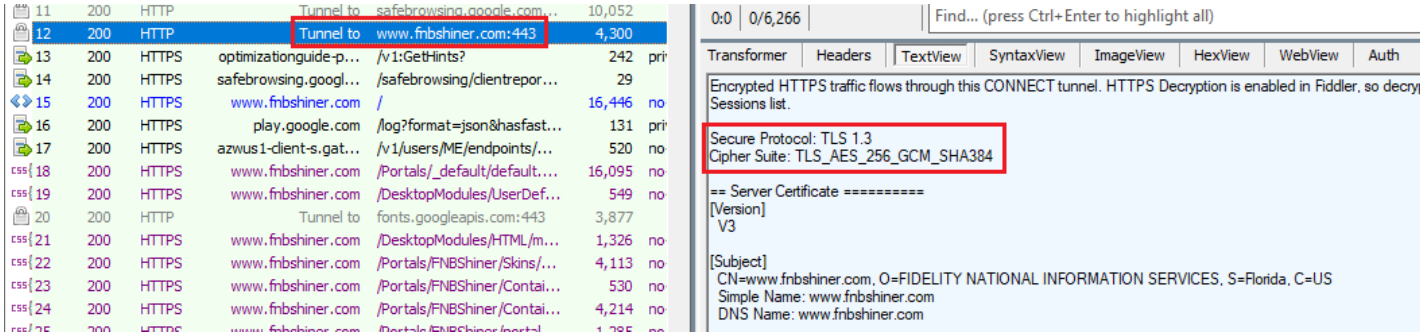
	 https://www.fnbshiner.com/

The Transport Layer Security (TLS) Protocol Version 1.3

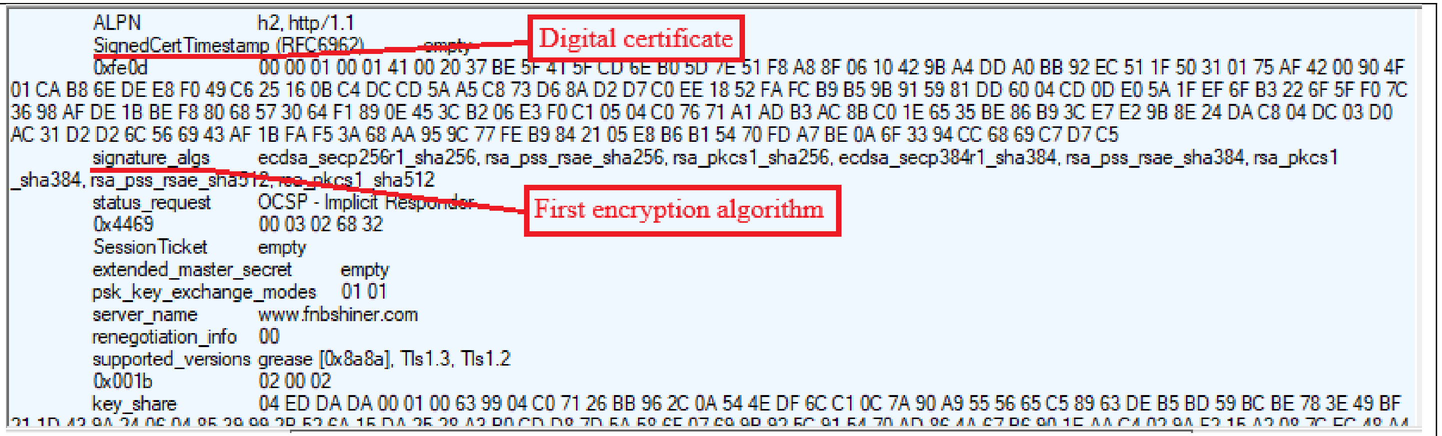
Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

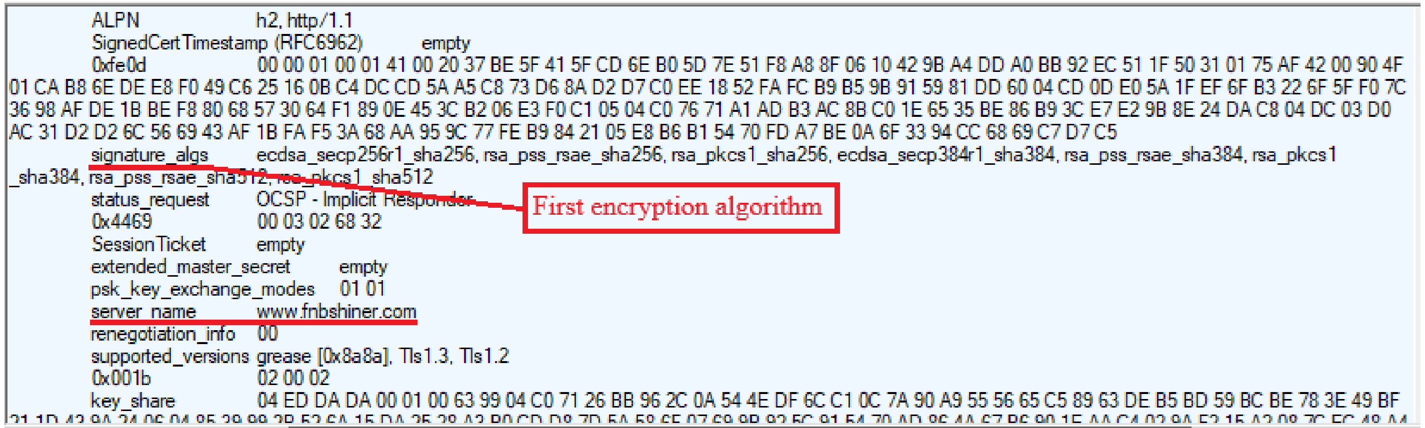
https://datatracker.ietf.org/doc/html/rfc8446



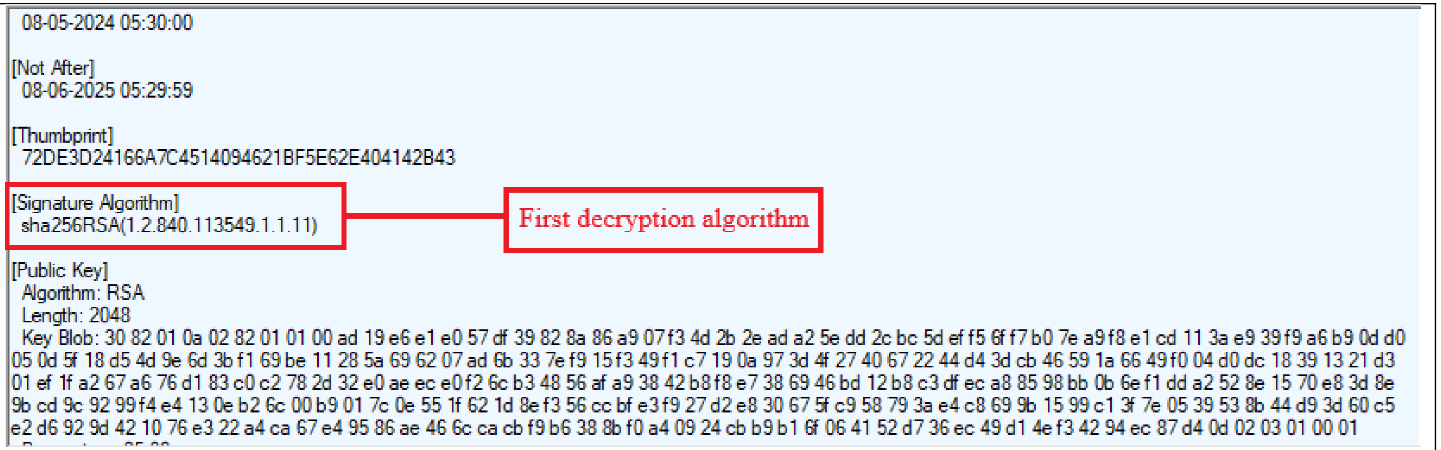
Source: Fiddler Capture



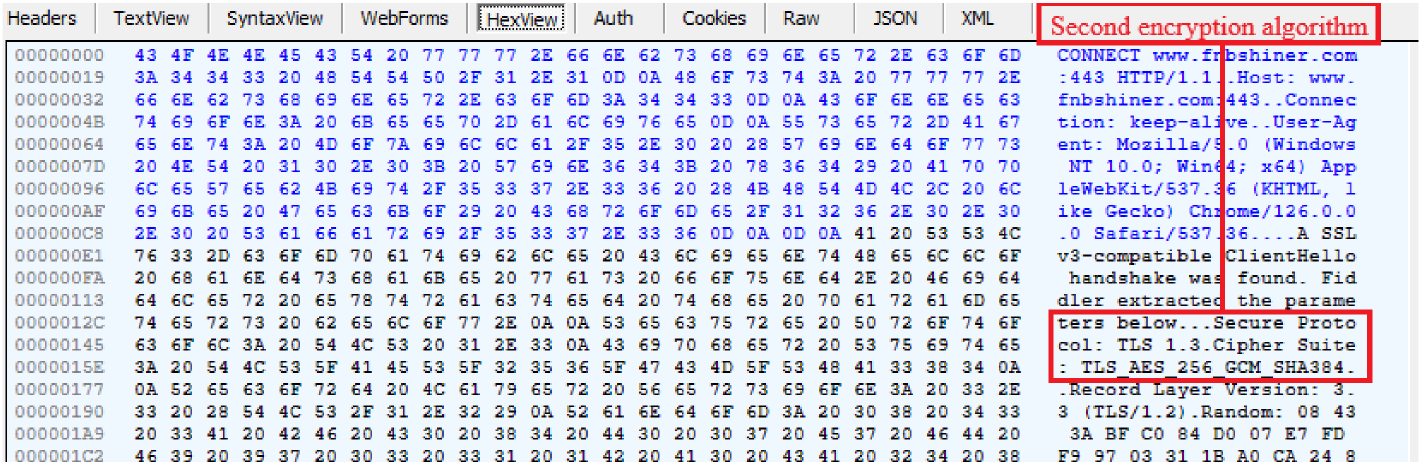
Source: Fiddler Capture



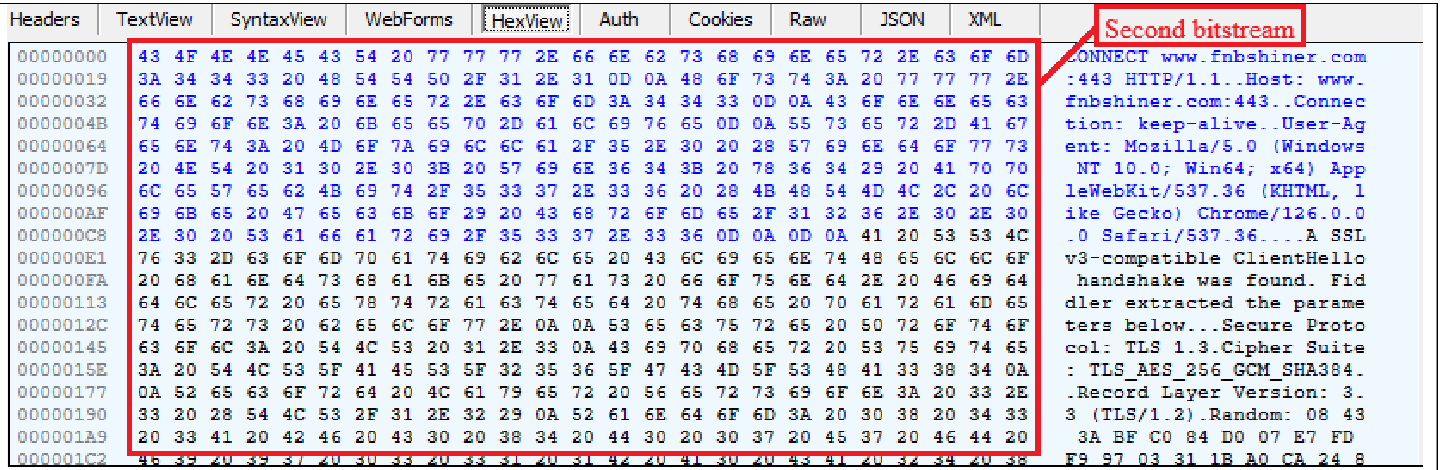
Source: Fiddler Capture



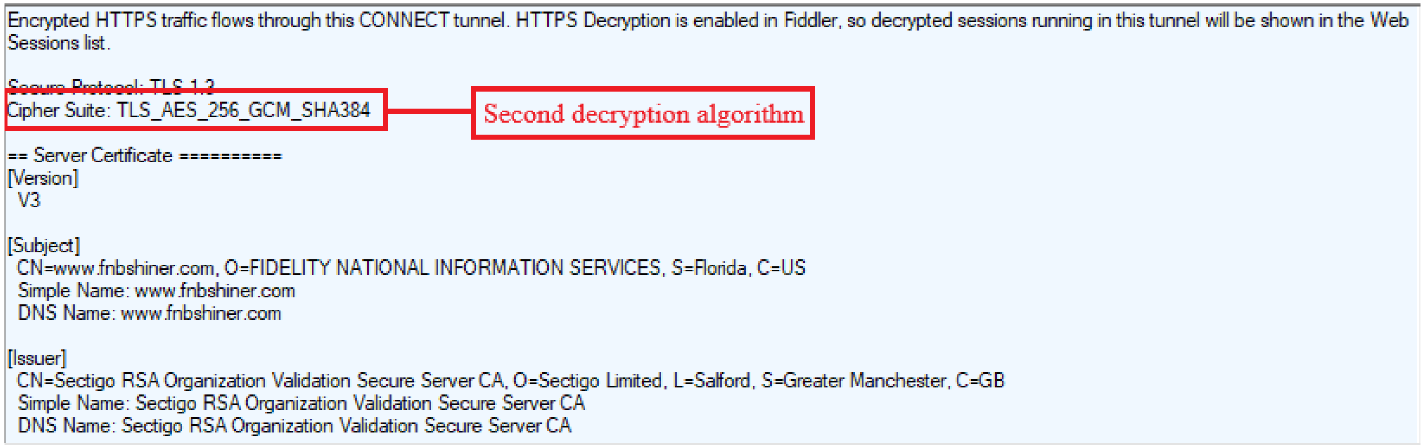
Source: Fiddler Capture



Source: Fiddler Capture

Source: Fiddler Capture



Source: Fiddler Capture

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for

	encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm. As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server. Further, the AEAD encrypted message comprises a ciphertext (e.g., encrypted ciphertext after the encryption by the second encryption algorithm), nonce (e.g., associating second decryption algo), key and associated data. The maximum length of nonce is a cipher suit specific element. The nonce and associated data are utilized in decryption of the AEAD encrypted message.

	5. Record Protocol The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients. TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see Appendix D.4). https://datatracker.ietf.org/doc/html/rfc8446#section-1 **2. Protocol Overview** Negotiating encryption algorithm The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic. https://datatracker.ietf.org/doc/html/rfc8446

	TLS consists of two primary components: - A handshake protocol (Section 4) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack. Negotiating encryption algos - A record protocol (Section 5) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys. https://datatracker.ietf.org/doc/html/rfc8446

	5.1. Record Layer The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^14 bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2. Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that: - Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **5.2. Record Payload Protection** The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see Section 5.3), and the additional data input is the record header. I.e., `additional_data = TLSCiphertext.opaque_type ‖` `TLSCiphertext.legacy_record_version ‖` `TLSCiphertext.length` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case. https://datatracker.ietf.org/doc/html/rfc5116

	2.1. Authenticated Encryption The authenticated encryption operation has four inputs, each of which is an octet string: A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom. A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in Section 3.2, and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces. A plaintext P, which contains the data to be encrypted and authenticated. The associated data A, which contains the data to be authenticated, but not encrypted. https://datatracker.ietf.org/doc/html/rfc5116

	2.2. Authenticated Decryption Second decryption algorithm The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure). https://datatracker.ietf.org/doc/html/rfc5116 The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [RFC5116], Section 4). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows: https://datatracker.ietf.org/doc/html/rfc8446#section-1 This specification defines the following cipher suites for use with TLS 1.3. | Description | Value | | TLS_AES_128_GCM_SHA256 | {0x13,0x01} | | TLS_AES_256_GCM_SHA384 | {0x13,0x02} | | TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} | | TLS_AES_128_CCM_SHA256 | {0x13,0x04} | | TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} | https://datatracker.ietf.org/doc/html/rfc8446#section-1

	All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection. https://datatracker.ietf.org/doc/html/rfc8446#section-1 **4.4. Authentication Messages** As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [sender]_handshake_traffic_secret. https://datatracker.ietf.org/doc/html/rfc8446#section-1

Figure 1 below shows the basic full TLS handshake:

```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

Digital Content

https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2. Certificate This message conveys the endpoint's certificate chain to the peer. The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK). The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.2.3. Client Certificate Selection The following rules apply to certificates sent by the client: - The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]). - If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs. - The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS. - If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.4.3. Certificate Verify This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message. Structure of this message: `struct {` `   SignatureScheme algorithm;` `   opaque signature<0..2^16-1>;` `} CertificateVerify;` The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely: https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	4.3.2. Certificate Request A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions. Structure of this message: `struct {` `    opaque certificate_request_context<0..2^8-1>;` `    Extension extensions<2..2^16-1>;` `} CertificateRequest;` https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

	certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages. extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions. https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2

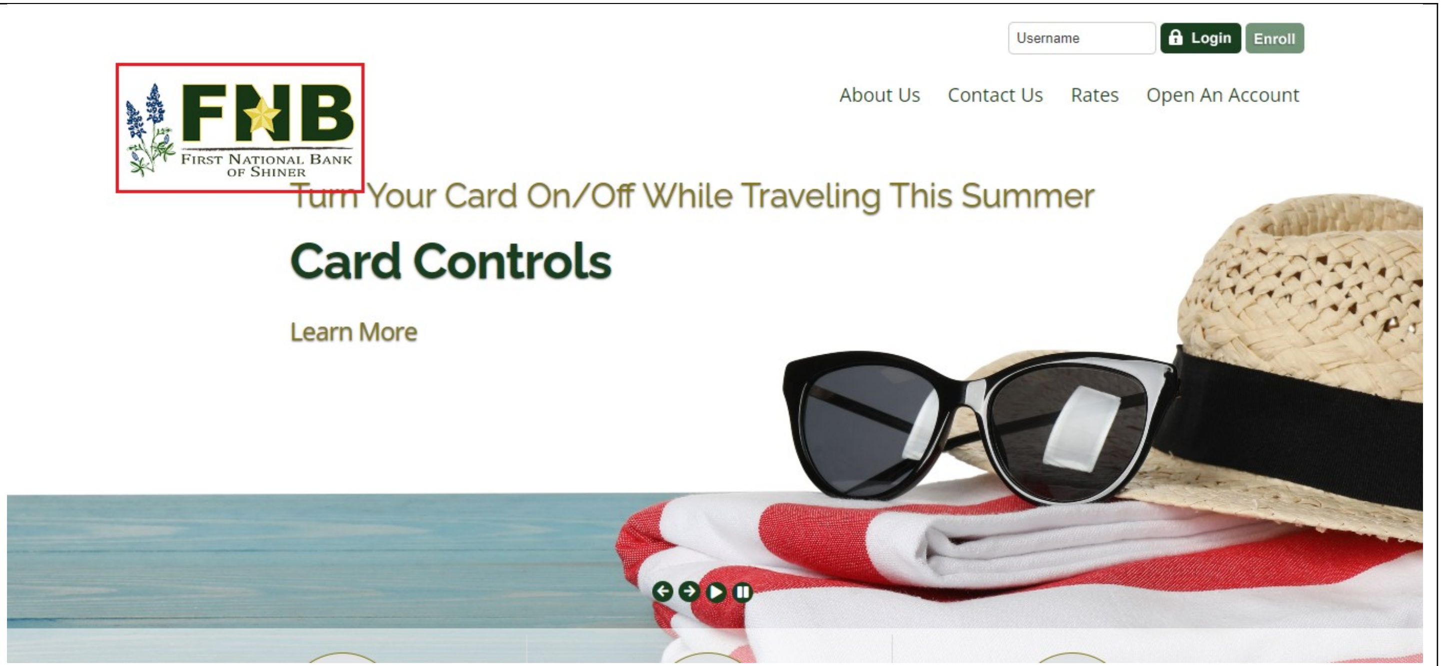
	- RSASSA-PSS signature schemes are defined in Section 4.2.3. - The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello. - The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates. https://datatracker.ietf.org/doc/html/rfc8446#section-1 If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message. In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages. https://datatracker.ietf.org/doc/html/rfc8446#section-1

	4.2.3. Signature Algorithms TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2). The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension. https://datatracker.ietf.org/doc/html/rfc8446#section-1

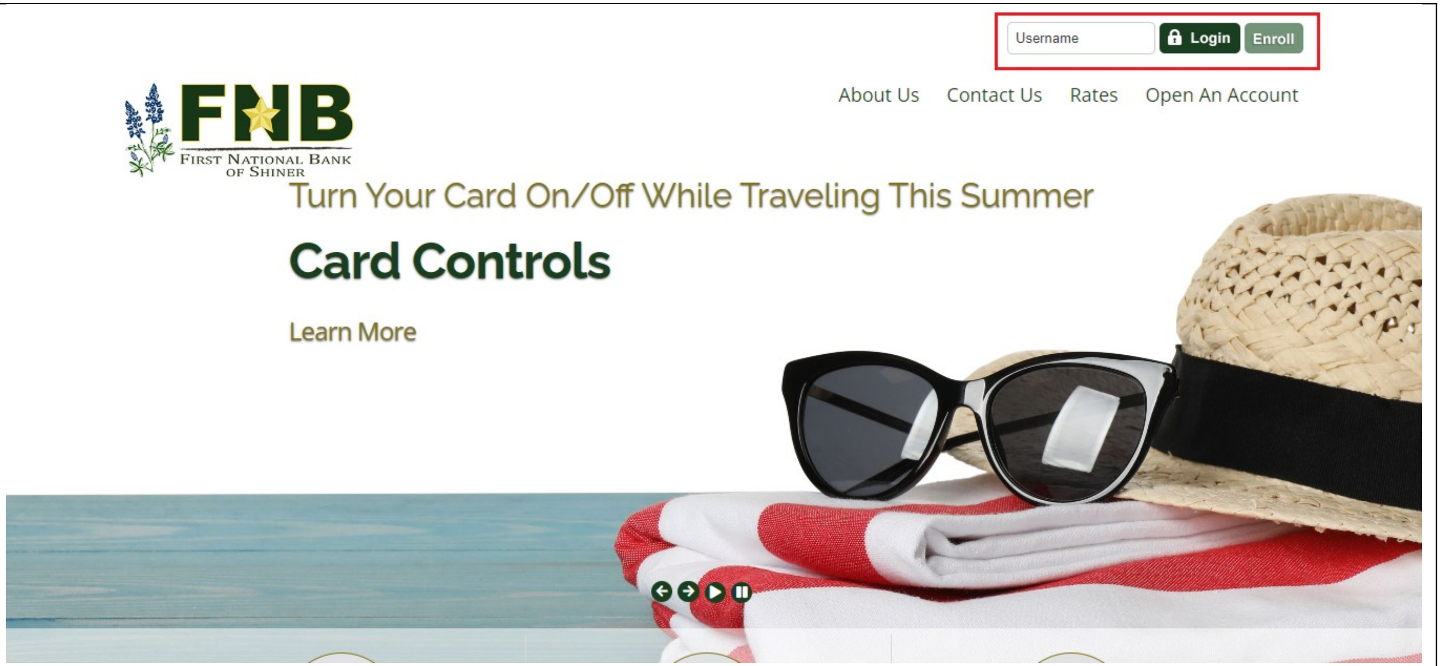
	The "extension_data" field of these extensions contains a SignatureSchemeList value: `enum {` `    /* RSASSA-PKCS1-v1_5 algorithms */` `    rsa_pkcs1_sha256(0x0401),` `    rsa_pkcs1_sha384(0x0501),` `    rsa_pkcs1_sha512(0x0601),` `    /* ECDSA algorithms */` `    ecdsa_secp256r1_sha256(0x0403),` `    ecdsa_secp384r1_sha384(0x0503),` `    ecdsa_secp521r1_sha512(0x0603),` `    /* RSASSA-PSS algorithms with public key OID rsaEncryption */` `    rsa_pss_rsae_sha256(0x0804),` `    rsa_pss_rsae_sha384(0x0805),` `    rsa_pss_rsae_sha512(0x0806),` `    /* EdDSA algorithms */` `    ed25519(0x0807),` `    ed448(0x0808),` `    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */` `    rsa_pss_pss_sha256(0x0809),` `    rsa_pss_pss_sha384(0x080a),` `    rsa_pss_pss_sha512(0x080b),` https://datatracker.ietf.org/doc/html/rfc8446#section-1 As shown below, the receiving party will be able to decrypt the encrypted message with the provided signature decryption algorithm information i.e., SHA-256 RSA decryption algorithm.

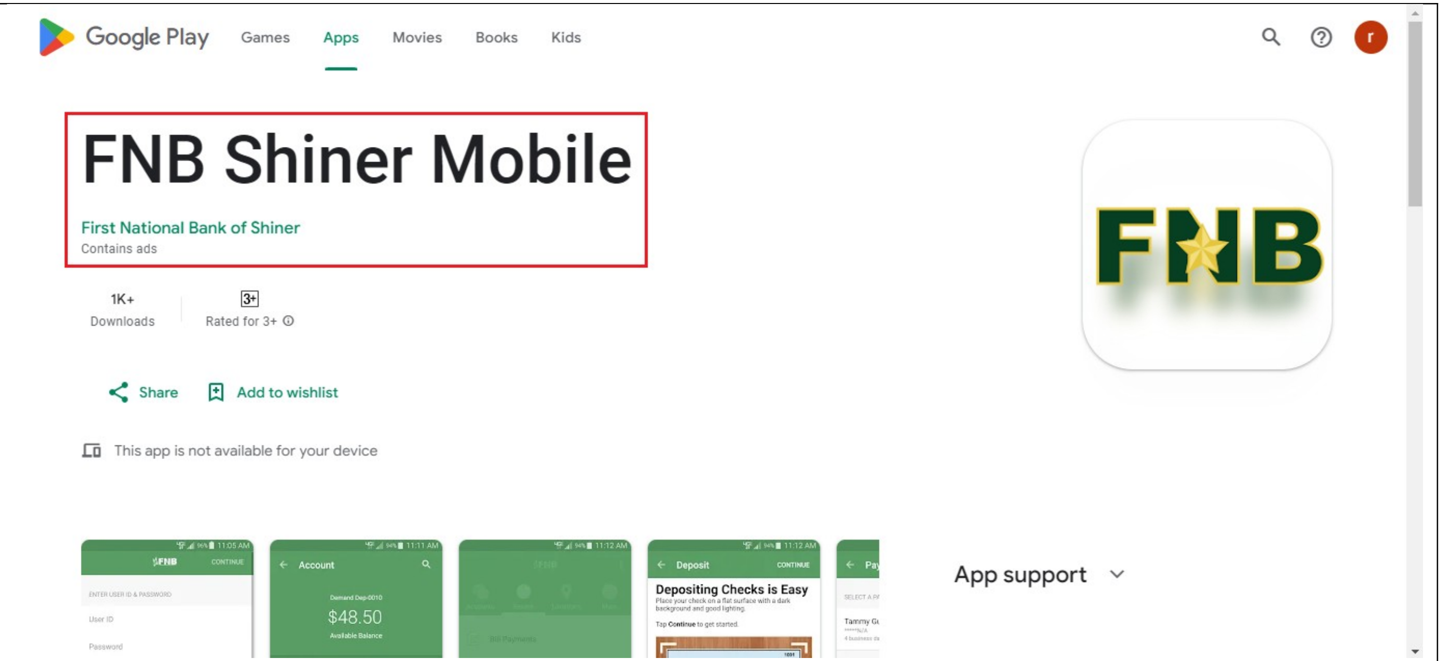
	We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q, compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de = 1 \pmod{\varphi(n)}$. we know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$. Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is $\boxed{c = m^e \bmod n.}$ First encryption To decrypt, we compute $c^d \bmod n$ to obtain $c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$ The result of Exercise 3.13 tells us that $\boxed{m \equiv m^{1+k\varphi(n)} \pmod{n},}$ First decryption https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf
39. The software system or computer program of claim 38, wherein the decrypting is done using a key associated with each decryption algorithm.	The standard practices the method such that the decrypting is done using a key (e.g., decryption key) associated with each decryption algorithm (e.g., signature decryption algorithm such as SHA-256RSA, etc., and AEAD decryption algorithm such as TLS_AES_256_GCM_SHA384, etc.).

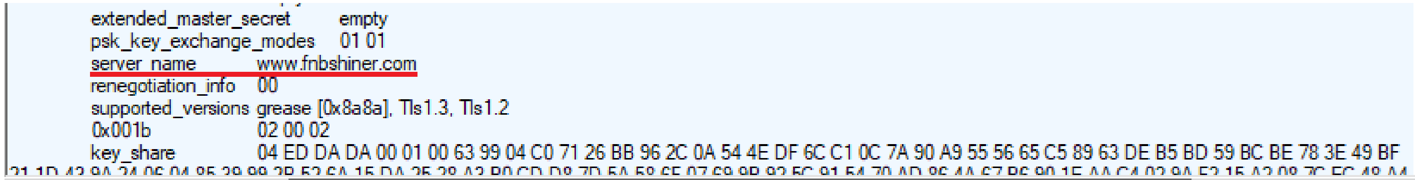
<table>
<tr>
<td></td>
<td>


https://www.fnbshiner.com/
</td>
</tr>
</table>

	 https://www.fnbshiner.com/



https://play.google.com/store/apps/details?id=com.mfoundry.mb.android.mb_113106833&hl=en_US



Source: Fiddler Capture

As shown below, the signature decryption algorithm utilizes a private key for a first decryption and the AEAD decryption algorithm uses a key K. Both the decryption techniques are decrypting using their respective associated keys.

	The "extension_data" field of these extensions contains a SignatureSchemeList value: `enum {` `/* RSASSA-PKCS1-v1_5 algorithms */` `rsa_pkcs1_sha256(0x0401),` `rsa_pkcs1_sha384(0x0501),` `rsa_pkcs1_sha512(0x0601),` `/* ECDSA algorithms */` `ecdsa_secp256r1_sha256(0x0403),` `ecdsa_secp384r1_sha384(0x0503),` `ecdsa_secp521r1_sha512(0x0603),` `/* RSASSA-PSS algorithms with public key OID rsaEncryption */` `rsa_pss_rsae_sha256(0x0804),` `rsa_pss_rsae_sha384(0x0805),` `rsa_pss_rsae_sha512(0x0806),` `/* EdDSA algorithms */` `ed25519(0x0807),` `ed448(0x0808),` `/* RSASSA-PSS algorithms with public key OID RSASSA-PSS */` `rsa_pss_pss_sha256(0x0809),` `rsa_pss_pss_sha384(0x080a),` `rsa_pss_pss_sha512(0x080b),` https://datatracker.ietf.org/doc/html/rfc8446#section-1

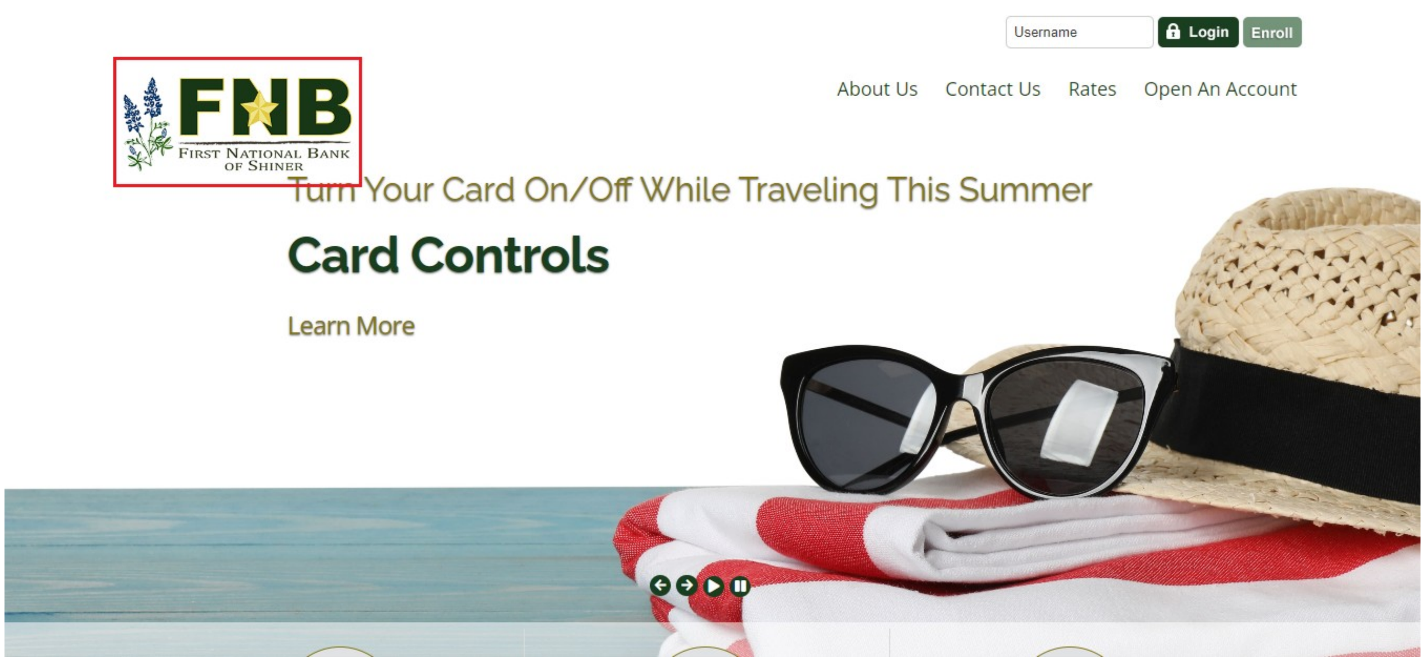
	There is also a decryption function D that takes a ciphertext and a decryption key K_D, and reproduces the plaintext message. $D(C, K_D) = P$ In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster. In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B's encryption key, encrypt a message with it, and send the result to person B. Only someone with B's decryption key, namely only B, can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

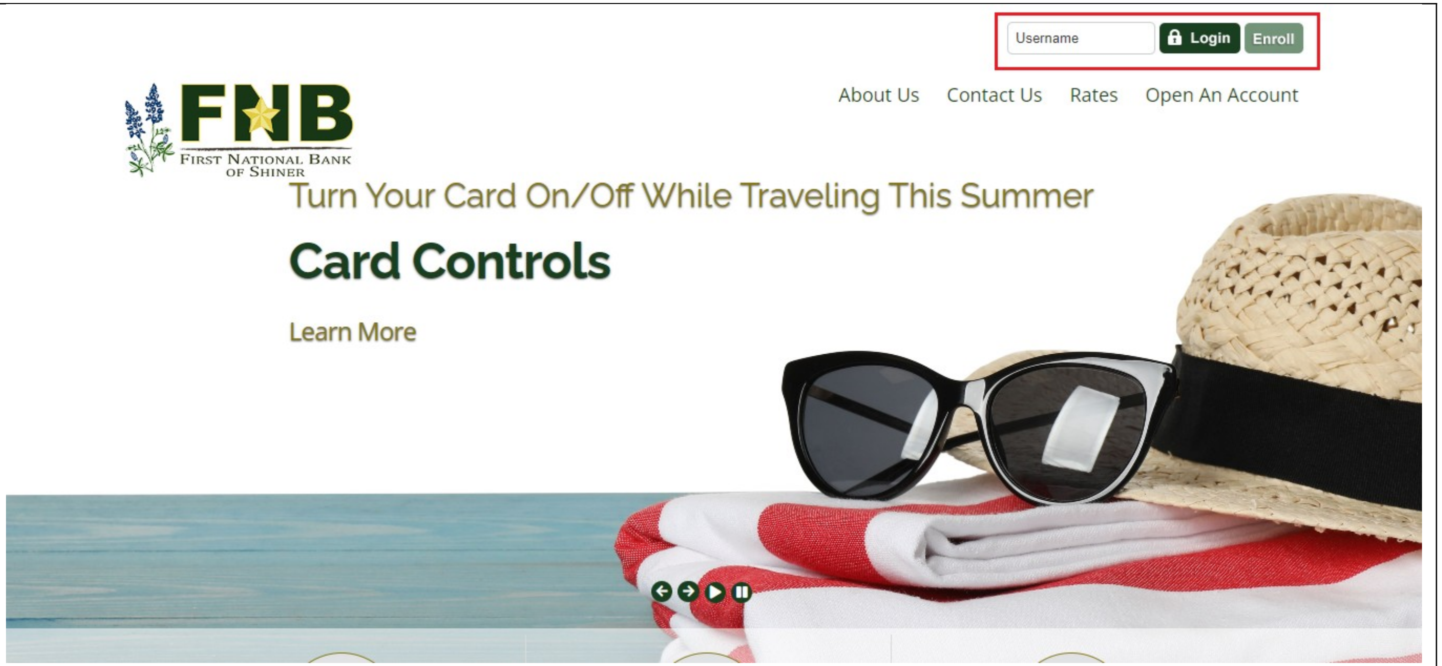
	We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q, compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de = 1 \pmod{\varphi(n)}$. we know that $de-1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$. Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is $\boxed{c = m^e \bmod n.}$ First encryption To decrypt, we compute $c^d \bmod n$ to obtain $c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$ The result of Exercise 3.13 tells us that $\boxed{m \equiv m^{1+k\varphi(n)} \pmod{n},}$ First decryption https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

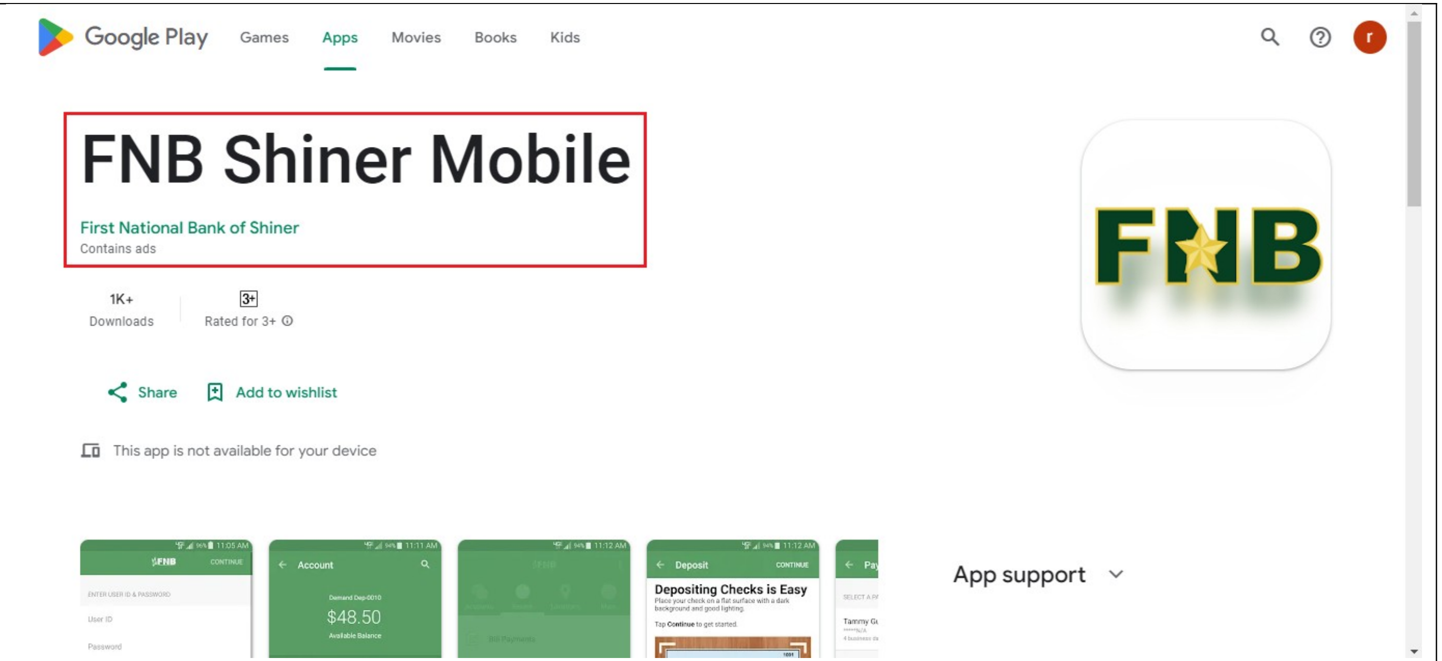
	The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	This specification defines the following cipher suites for use with TLS 1.3. ``` +------------------------------+-------------+ | Description | Value | +------------------------------+-------------+ | TLS_AES_128_GCM_SHA256 | {0x13,0x01} | | | | | TLS_AES_256_GCM_SHA384 | {0x13,0x02} | | | | | TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} | | | | | TLS_AES_128_CCM_SHA256 | {0x13,0x04} | | | | | TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} | +------------------------------+-------------+ ``` https://datatracker.ietf.org/doc/html/rfc8446#section-1

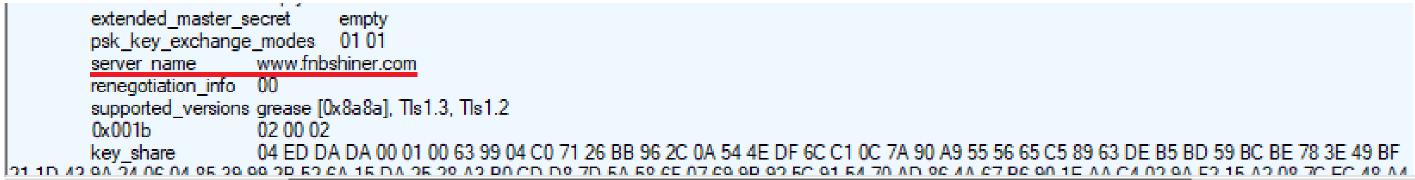
	2.2. Authenticated Decryption The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure). https://datatracker.ietf.org/doc/html/rfc5116 The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1
40. The software system or computer program of claim 39, wherein the key is resident in hardware of the target unit or the key is retrieved from a	The standard utilized by the accused instrumentality practices the method such that the key is resident in hardware (e.g., stored in a memory storage of the server such as a database, RAM, etc.) of the target unit (e.g., server of the accused instrumentality) or the key is retrieved from a server.

server.	 https://www.fnbshiner.com/

	 https://www.fnbshiner.com/



https://play.google.com/store/apps/details?id=com.mfoundry.mb.android.mb_113106833&hl=en_US



extended_master_secret empty
psk_key_exchange_modes 01 01
server_name www.fnbshiner.com
renegotiation_info 00
supported_versions grease [0x8a8a], Tls1.3, Tls1.2
0x001b 02 00 02
key_share 04 ED DA DA 00 01 00 63 99 04 C0 71 26 BB 96 2C 0A 54 4E DF 6C C1 0C 7A 90 A9 55 56 65 C5 89 63 DE B5 BD 59 BC BE 78 3E 49 BF

Source: Fiddler Capture

	Tech Accelerator **Server hardware guide: Architecture, products and management** **3. Random access memory** RAM is the main type of memory in a computing system. RAM holds the software instructions and data needed by the processor, along with any output from the processor, such as data to be moved to a storage device. Thus, RAM works very closely with the processor and must match the processor's incredible speed and performance. This kind of fast memory is usually termed dynamic RAM, and several DRAM variations are available for servers. https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms

Tech Accelerator

Server hardware guide: Architecture, products and management

4. Hard disk drive

This hardware is responsible for reading, writing and positioning of the hard disk, which is one technology for data storage on server hardware. Developed at IBM in 1953, the hard disk drive (HDD) has evolved over time from the size of a refrigerator to the standard 2.5-inch and 3.5-inch form factors.

https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms

As shown below, the server comprises a memory storage to store messages for establishing secure TLS communication. the standard discloses multiple signature encryption algorithms for a first encryption and multiple AEAD encryption algorithms for the second encryption. A signature decryption algorithm utilizes a private key for decrypting the first bitstream encrypted with the signature encryption and an AEAD decryption algorithm uses a key K for decrypting the second bitstream encrypted with the AEAD encryption. Both the decryption techniques are decrypting using their respective associated keys. A server must have a storage to store information pertaining to these algorithms and their corresponding keys such as private key, Key K, etc., to establish secure TLS communication with a client.

	Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT. https://datatracker.ietf.org/doc/html/rfc8446#

	The "extension_data" field of these extensions contains a SignatureSchemeList value: `enum {` `    /* RSASSA-PKCS1-v1_5 algorithms */` `    rsa_pkcs1_sha256(0x0401),` `    rsa_pkcs1_sha384(0x0501),` `    rsa_pkcs1_sha512(0x0601),` `    /* ECDSA algorithms */` `    ecdsa_secp256r1_sha256(0x0403),` `    ecdsa_secp384r1_sha384(0x0503),` `    ecdsa_secp521r1_sha512(0x0603),` `    /* RSASSA-PSS algorithms with public key OID rsaEncryption */` `    rsa_pss_rsae_sha256(0x0804),` `    rsa_pss_rsae_sha384(0x0805),` `    rsa_pss_rsae_sha512(0x0806),` `    /* EdDSA algorithms */` `    ed25519(0x0807),` `    ed448(0x0808),` `    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */` `    rsa_pss_pss_sha256(0x0809),` `    rsa_pss_pss_sha384(0x080a),` `    rsa_pss_pss_sha512(0x080b),` https://datatracker.ietf.org/doc/html/rfc8446#section-1

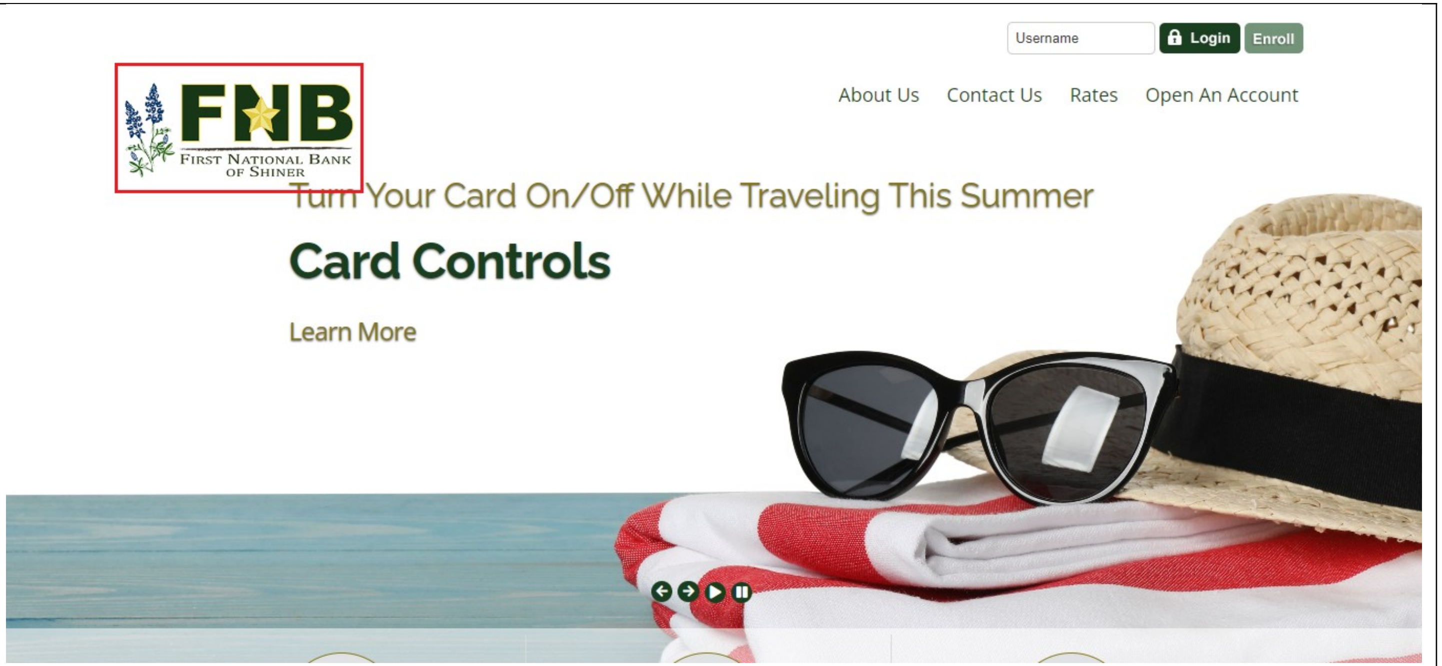
	There is also a decryption function D that takes a ciphertext and a decryption key K_D, and reproduces the plaintext message. $D(C, K_D) = P$ In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster. In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B's encryption key, encrypt a message with it, and send the result to person B. Only someone with B's decryption key, namely only B, can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q, compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de = 1 \pmod{\varphi(n)}$. we know that $de-1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$. Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is $\boxed{c = m^e \bmod n.}$ First encryption To decrypt, we compute $c^d \bmod n$ to obtain $c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$ The result of Exercise 3.13 tells us that $\boxed{m \equiv m^{1+k\varphi(n)} \pmod{n},}$ First decryption https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

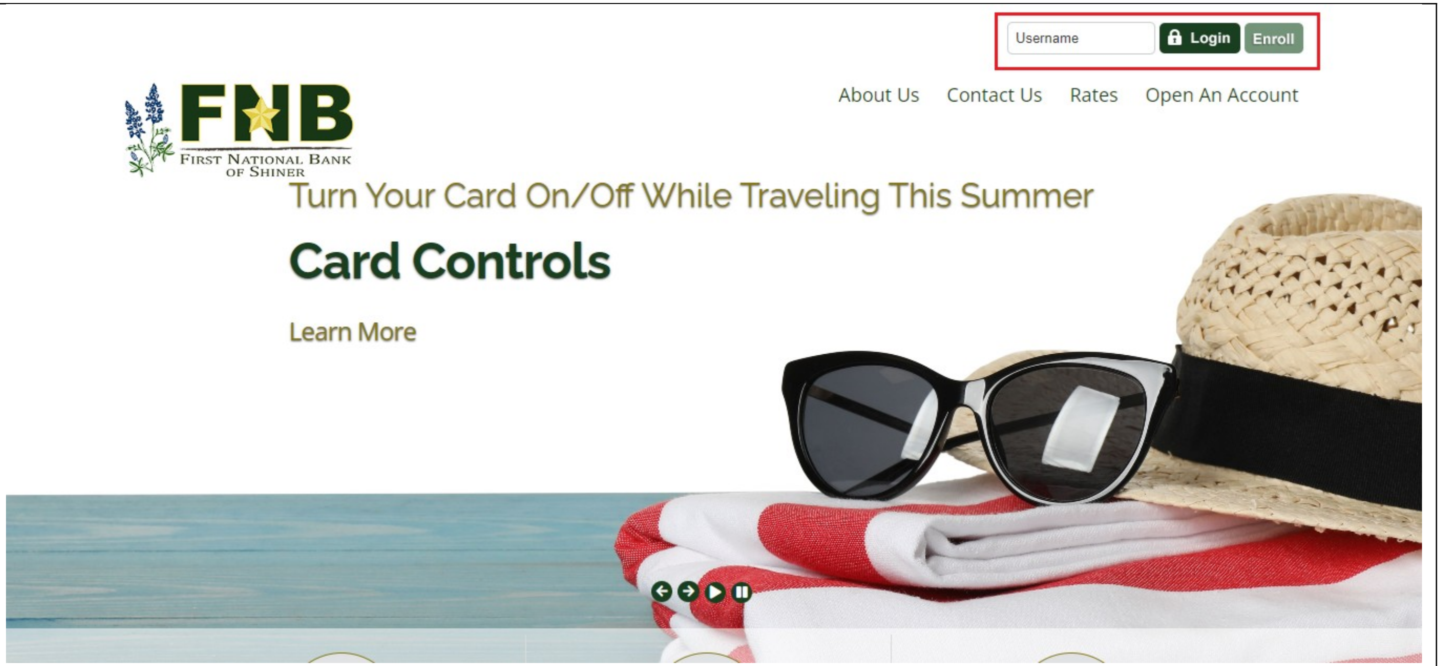
	The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

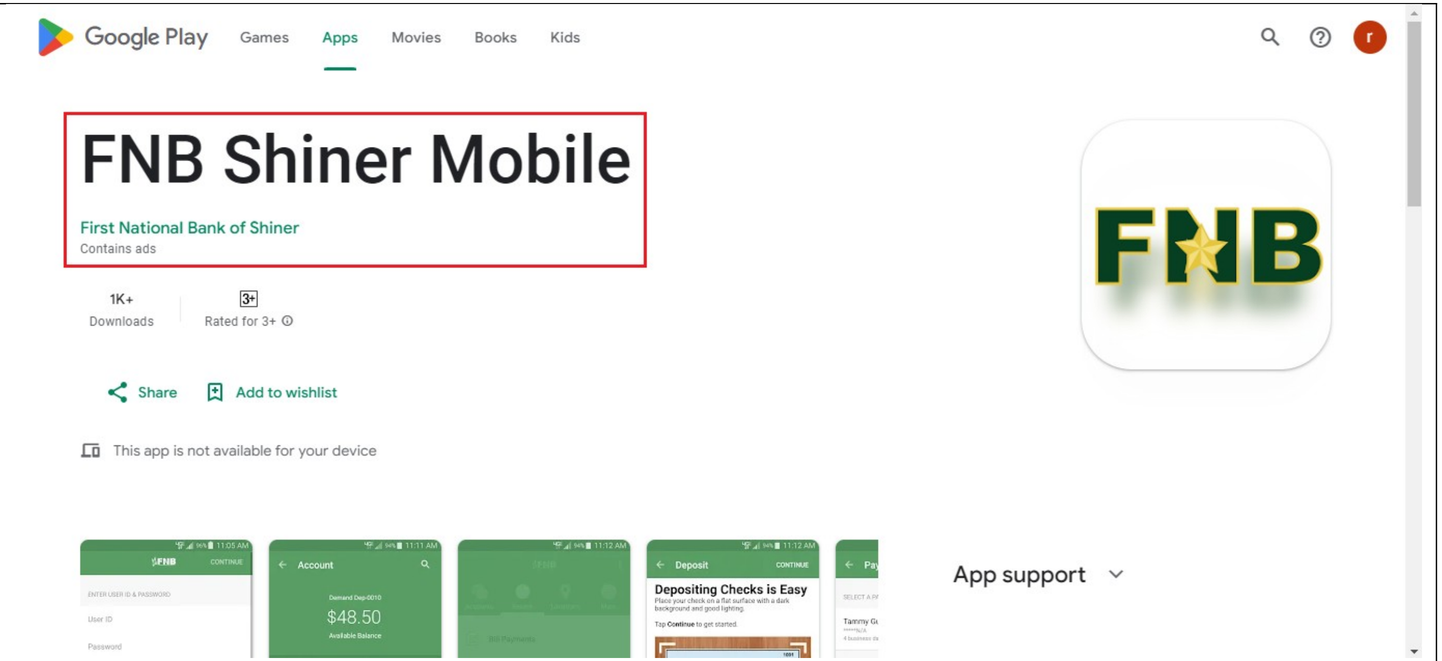
	This specification defines the following cipher suites for use with TLS 1.3. | Description | Value | | TLS_AES_128_GCM_SHA256 | {0x13,0x01} | | TLS_AES_256_GCM_SHA384 | {0x13,0x02} | | TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} | | TLS_AES_128_CCM_SHA256 | {0x13,0x04} | | TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} | https://datatracker.ietf.org/doc/html/rfc8446#section-1

	2.2. Authenticated Decryption The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure). https://datatracker.ietf.org/doc/html/rfc5116 The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1
41. The software system or computer program of claim 40, wherein the key is contained in a key data structure.	The standard utilized by the accused instrumentality practices the method such that the key (e.g., private key, Key K, etc.) is contained in a key data structure (e.g., data structure).

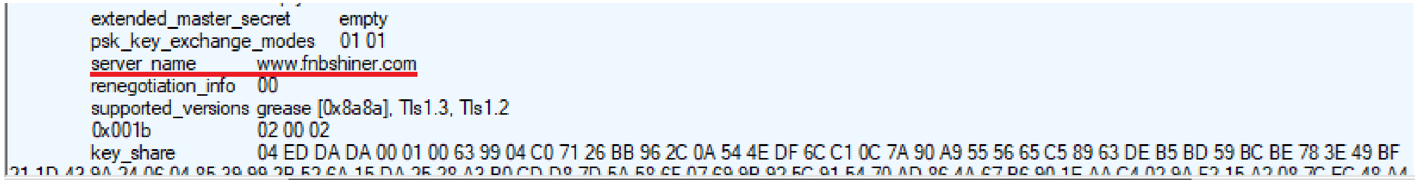
<table>
<tr>
<td></td>
<td>


https://www.fnbshiner.com/
</td>
</tr>
</table>

	 https://www.fnbshiner.com/



https://play.google.com/store/apps/details?id=com.mfoundry.mb.android.mb_113106833&hl=en_US



Source: Fiddler Capture

The accused instrumentality utilizes a server to establish a secure TLS communication with a client. The server must comprise a memory storage and store data according to a data structure to implement the standard efficiently.

	Tech Accelerator **Server hardware guide: Architecture, products and management** **3. Random access memory** RAM is the main type of memory in a computing system. RAM holds the software instructions and data needed by the processor, along with any output from the processor, such as data to be moved to a storage device. Thus, RAM works very closely with the processor and must match the processor's incredible speed and performance. This kind of fast memory is usually termed dynamic RAM, and several DRAM variations are available for servers. https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms

	Tech Accelerator **Server hardware guide: Architecture, products and management** **4. Hard disk drive** This hardware is responsible for reading, writing and positioning of the hard disk, which is one technology for data storage on server hardware. Developed at IBM in 1953, the hard disk drive (HDD) has evolved over time from the size of a refrigerator to the standard 2.5-inch and 3.5-inch form factors. https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms

	A data structure is a specialized format for organizing, processing, retrieving and storing data. There are several basic and advanced types of data structures, all designed to arrange data to suit a specific purpose. Data structures make it easy for users to access and work with the data they need in appropriate ways. Most importantly, data structures frame the organization of information so that machines and humans can better understand it. In computer science and computer programming, a data structure may be selected or designed to store data for the purpose of using it with various algorithms. In some cases, the algorithm's basic operations are tightly coupled to the data structure's design. Each data structure contains information about the data values, relationships between the data and -- in some cases -- functions that can be applied to the data. https://www.techtarget.com/searchdatamanagement/definition/data-structure As shown below, the server comprises a memory storage to store messages for establishing secure TLS communication. the standard discloses multiple signature encryption algorithms for a first encryption and multiple AEAD encryption algorithms for the second encryption. A signature decryption algorithm utilizes a private key for decrypting the first bitstream encrypted with the signature encryption and an AEAD decryption algorithm uses a key K for decrypting the second bitstream encrypted with the AEAD encryption. Both the decryption techniques are decrypting using their respective associated keys. A server must have a storage to store information pertaining to these algorithms and their corresponding keys such as private key, Key K, etc., to establish secure TLS communication with a client.

	Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT. https://datatracker.ietf.org/doc/html/rfc8446#

	The "extension_data" field of these extensions contains a SignatureSchemeList value: `enum {` `    /* RSASSA-PKCS1-v1_5 algorithms */` `    rsa_pkcs1_sha256(0x0401),` `    rsa_pkcs1_sha384(0x0501),` `    rsa_pkcs1_sha512(0x0601),` `    /* ECDSA algorithms */` `    ecdsa_secp256r1_sha256(0x0403),` `    ecdsa_secp384r1_sha384(0x0503),` `    ecdsa_secp521r1_sha512(0x0603),` `    /* RSASSA-PSS algorithms with public key OID rsaEncryption */` `    rsa_pss_rsae_sha256(0x0804),` `    rsa_pss_rsae_sha384(0x0805),` `    rsa_pss_rsae_sha512(0x0806),` `    /* EdDSA algorithms */` `    ed25519(0x0807),` `    ed448(0x0808),` `    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */` `    rsa_pss_pss_sha256(0x0809),` `    rsa_pss_pss_sha384(0x080a),` `    rsa_pss_pss_sha512(0x080b),` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	There is also a decryption function D that takes a ciphertext and a decryption key K_D, and reproduces the plaintext message. $D(C, K_D) = P$ In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster. In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B's encryption key, encrypt a message with it, and send the result to person B. Only someone with B's decryption key, namely only B, can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q, compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de = 1 \pmod{\varphi(n)}$. we know that $de-1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$. Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is $\boxed{c = m^e \bmod n.}$ First encryption To decrypt, we compute $c^d \bmod n$ to obtain $c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$ The result of Exercise 3.13 tells us that $\boxed{m \equiv m^{1+k\varphi(n)} \pmod{n},}$ First decryption https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	This specification defines the following cipher suites for use with TLS 1.3. ``` +------------------------------+-------------+ | Description	Value | +------------------------------+-------------+ | TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05} | +------------------------------+-------------+ ``` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	2.2. Authenticated Decryption The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure). https://datatracker.ietf.org/doc/html/rfc5116 The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1
47. The software system or computer program of claim 39, wherein each encryption algorithm is a symmetric key system or an	The standard practices the method such that each encryption algorithm (e.g., signature encryption algorithm i.e., SHA256RSA, etc., and AEAD encryption algorithm i.e., TLS_AES_256_GCM_SHA384, etc.) is a symmetric key system (e.g., AEAD encryption algorithm, etc.) or an asymmetric key system (e.g., signature encryption algorithm). As shown below, the server comprises a memory storage to store messages for

asymmetric key system.	establishing secure TLS communication. the standard discloses multiple signature encryption algorithms for a first encryption and multiple AEAD encryption algorithms for the second encryption. A signature decryption algorithm utilizes a private key for decrypting the first bitstream encrypted with the signature encryption and an AEAD decryption algorithm uses a key K for decrypting the second bitstream encrypted with the AEAD encryption. The standard defines the signature encryption algorithm as an asymmetric cryptography algorithm and the AEAD encryption algorithm as the symmetric cryptography algorithm. Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT. https://datatracker.ietf.org/doc/html/rfc8446# Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated. Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], the Elliptic Curve Digital Signature Algorithm (ECDSA) [ECDSA], or the Edwards-Curve Digital Signature Algorithm (EdDSA) [RFC8032]) or a symmetric pre-shared key (PSK). https://datatracker.ietf.org/doc/html/rfc8446#section-4

	cipher_suites: A list of the symmetric cipher options supported by the client, specifically the record protection algorithm (including secret key length) and a hash to be used with HKDF, in descending order of client preference. Values are defined in Appendix B.4. If the list contains cipher suites that the server does not recognize, support, or wish to use, the server MUST ignore those cipher suites and process the remaining ones as usual. If the client is attempting a PSK key establishment, it SHOULD advertise at least one cipher suite indicating a Hash associated with the PSK. https://datatracker.ietf.org/doc/html/rfc8446#section-4

	The "extension_data" field of these extensions contains a SignatureSchemeList value: ```enum { /* RSASSA-PKCS1-v1_5 algorithms */ rsa_pkcs1_sha256(0x0401), rsa_pkcs1_sha384(0x0501), rsa_pkcs1_sha512(0x0601), /* ECDSA algorithms */ ecdsa_secp256r1_sha256(0x0403), ecdsa_secp384r1_sha384(0x0503), ecdsa_secp521r1_sha512(0x0603), /* RSASSA-PSS algorithms with public key OID rsaEncryption */ rsa_pss_rsae_sha256(0x0804), rsa_pss_rsae_sha384(0x0805), rsa_pss_rsae_sha512(0x0806), /* EdDSA algorithms */ ed25519(0x0807), ed448(0x0808), /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */ rsa_pss_pss_sha256(0x0809), rsa_pss_pss_sha384(0x080a), rsa_pss_pss_sha512(0x080b),``` https://datatracker.ietf.org/doc/html/rfc8446#section-1

	There is also a decryption function D that takes a ciphertext and a decryption key K_D, and reproduces the plaintext message. $D(C, K_D) = P$ In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster. In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B's encryption key, encrypt a message with it, and send the result to person B. Only someone with B's decryption key, namely only B, can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

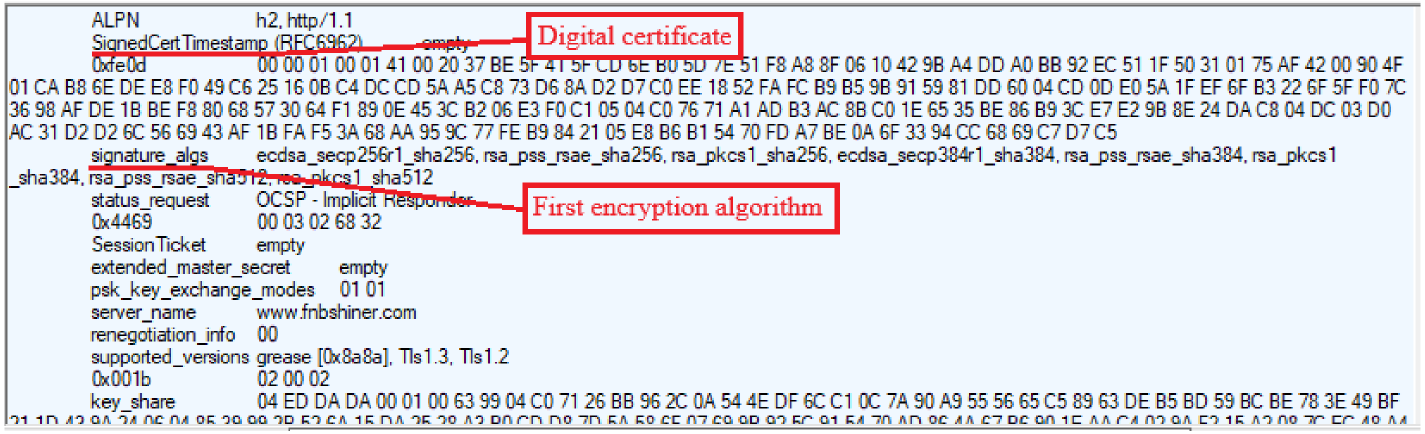
	The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions `MD5`, `SHA-1`, and `SHA-256`. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf

	This specification defines the following cipher suites for use with TLS 1.3. | Description | Value | | TLS_AES_128_GCM_SHA256 | {0x13,0x01} | | TLS_AES_256_GCM_SHA384 | {0x13,0x02} | | TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} | | TLS_AES_128_CCM_SHA256 | {0x13,0x04} | | TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} | https://datatracker.ietf.org/doc/html/rfc8446#section-1

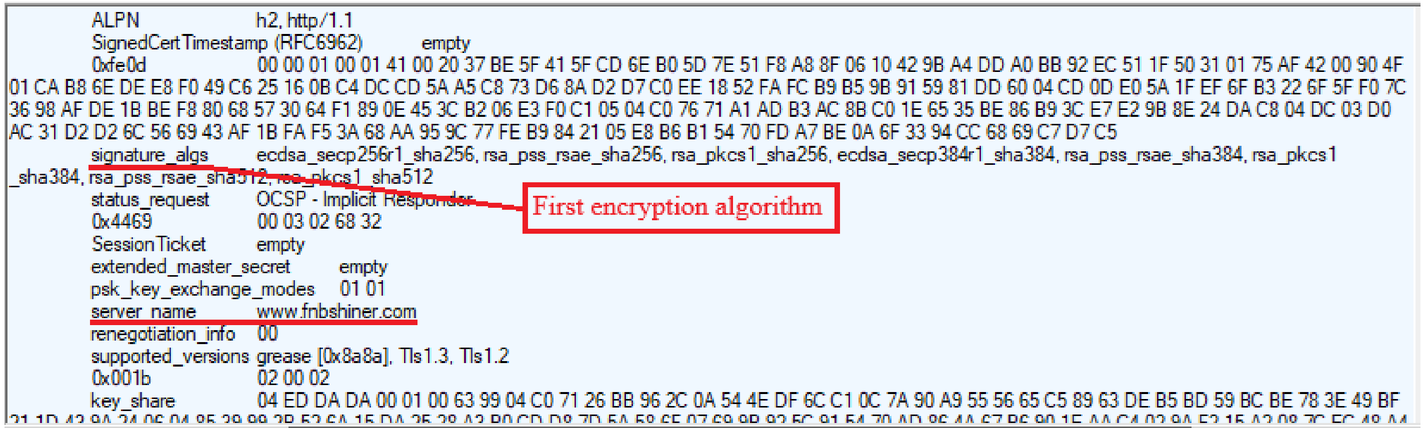
	The authenticated encryption operation has four inputs, each of which is an octet string: A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom. A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in Section 3.2, and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces. A plaintext P, which contains the data to be encrypted and authenticated. The associated data A, which contains the data to be authenticated, but not encrypted. https://datatracker.ietf.org/doc/html/rfc5116

	2.2. Authenticated Decryption The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure). https://datatracker.ietf.org/doc/html/rfc5116 The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. https://datatracker.ietf.org/doc/html/rfc8446#section-1
48. The software system or computer program of claim 39, further translatable for associating a first Message Authentication Code	The standard practices associating a first Message Authentication Code (MAC) (e.g., message authentication code with hashing function) or first digital signature with each encrypted bit stream (e.g., encrypted bit stream with the signature encryption algorithm i.e., SHA256RSA, etc., and encrypted bitstream with the AEAD encryption algorithm i.e., TLS_AES_256_GCM_SHA384, etc.). As shown below, the standard discloses a hashing function with each of the encryption

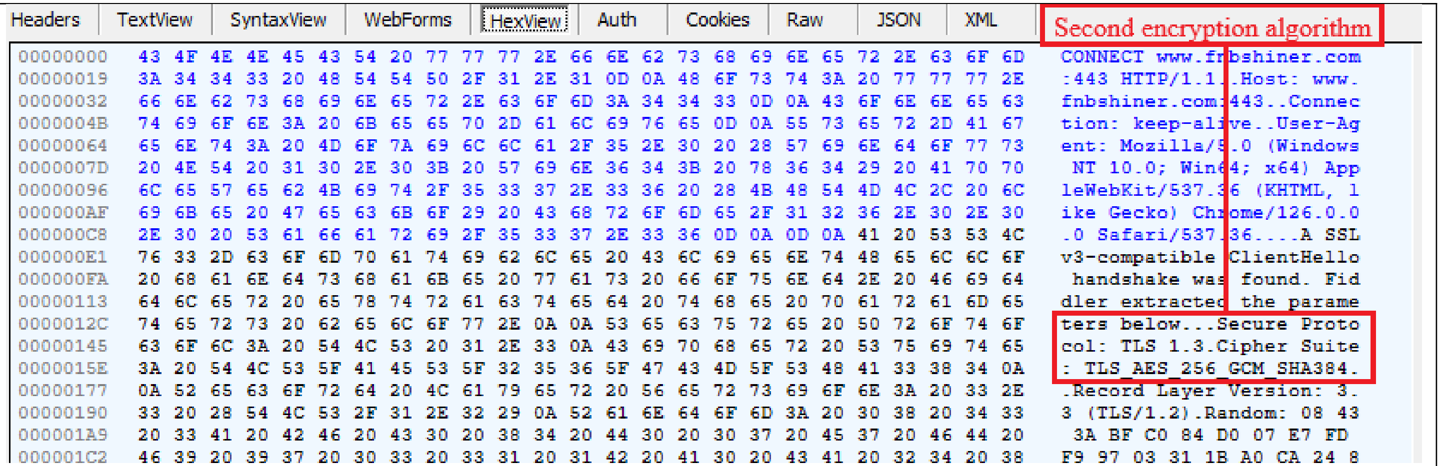
<table>
<tr>
<td>(MAC) or first digital signature with each encrypted bit stream.</td>
<td>algorithm. It performs a message authentication code with the utilized hashing function.



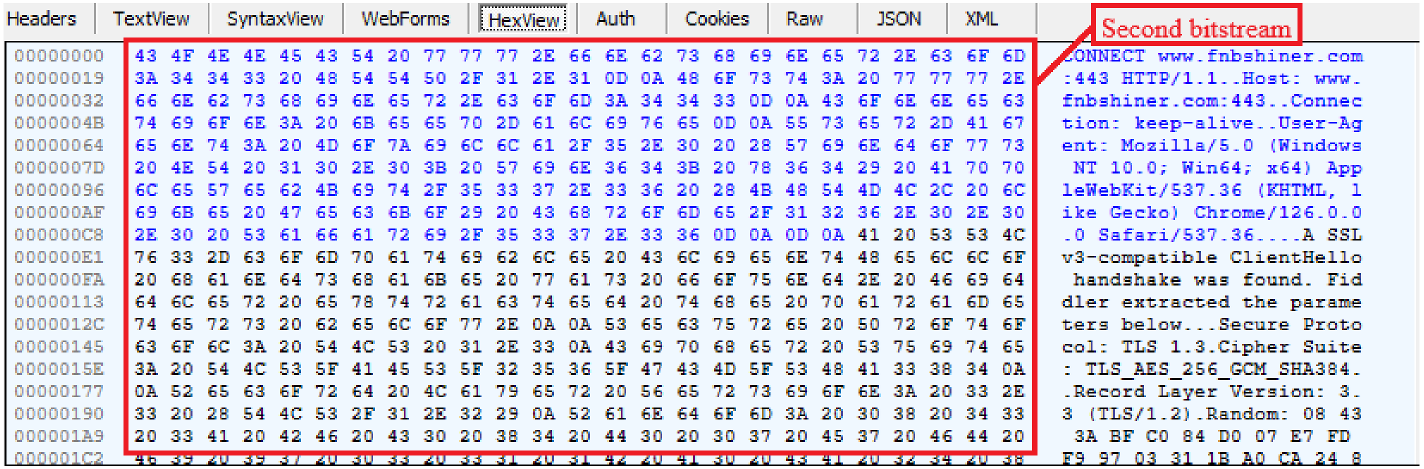
Source: Fiddler Capture



Source: Fiddler Capture</td>
</tr>
</table>



Source: Fiddler Capture



Source: Fiddler Capture

	The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m,h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property. https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf The list of supported symmetric encryption algorithms has been pruned of all algorithms that are considered legacy. Those that remain are all Authenticated Encryption with Associated Data (AEAD) algorithms. The cipher suite concept has been changed to separate the authentication and key exchange mechanisms from the record protection algorithm (including secret key length) and a hash to be used with both the key derivation function and handshake message authentication code (MAC). https://datatracker.ietf.org/doc/html/rfc8446#section-4